CYBR 435: Cyber Risk
Spring 2022

Lab Assignment #2: Buffer Overflows

- Name only: ____
- Release date: Feb 03, 2022 (Thursday), 2:00 pm
- Due date: Feb 10, 2022 (Thursday), 2:00pm
- Assignment should be **SUBMITTED on Blackboard before Due Date**. Other submission methods will NOT be accepted.
- **LATE Submission will NOT Be Accepted** on Blackboard since the submission link will be closed automatically after due date;
    - Additional submission for missing answer **will NOT Be Accepted**.
- It should be done INDIVIDUALLY; **Show ALL your work and evidence to support your answers**.
    - Answer only without evidence receives half credits.
- Total: 10 pts

Introduction

In this lab, you will learn how buffer overflows and other memory vulnerabilities are used to takeover vulnerable programs. The goal is to investigate the provided program and then figure out how to use it to gain shell access to systems.

In 1996 Aleph One wrote the canonical paper on smashing the stack. You should read this as it gives a detailed description of how stack smashing works. Today, many compilers and operating systems have implemented security features, which stop the attacks described in the paper. However, it still provides very relevant background for newer attacks and (specifically) this lab assignment.

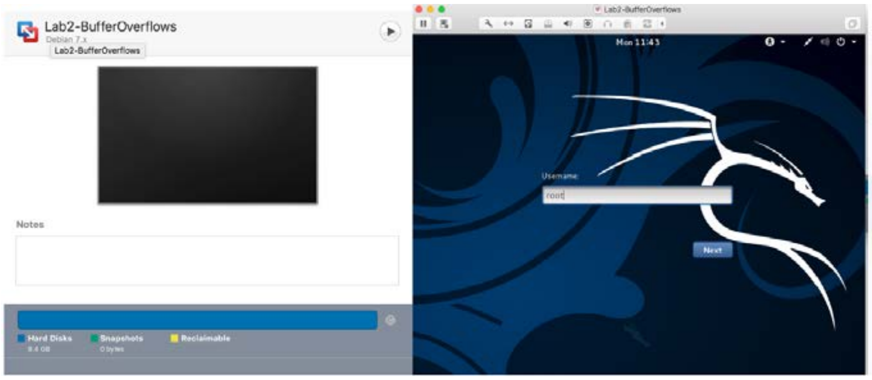
Aleph One: Smashing the Stack for Fun and Profit:
https://inst.eecs.berkeley.edu/~cs161/fa08/papers/stack_smashing.pdf

Another (long) description of Buffer Overflows is here:
https://www.win.tue.nl/~aeb/linux/hh/bof-eng.txt

Software Requirements

All required files and source code are packed in the provided Lab 2 virtual machine.
- The VMWare Software
  - https://www.vmware.com/
- The VirtualBox Software
  - https://www.virtualbox.org/wiki/Downloads
  - https://www.vmware.com/support/developer/ovf/
  - https://www.mylearning.be/2017/12/convert-a-vmware-fusion-virtual-machine-to-virtualbox-on-mac/
- The Kali Linux, Penetration Testing Distribution
- GDB: The GNU Project Debugger
- GCC, the GNU Compiler Collection
- C source file including BOF.c, createBadfile.c, and testShellCode.c

Starting the Virtual Machine
The Kali Linux VM has all the required files. Select the VM named Lab2-BufferOverflows for this lab.



Login the Kali Linux with the username and password [TBA in the class].

In the Kali Linux, you should be able to see a folder named Lab2-BufferOverflows. This folder contains all of the source code for the lab 2.

Setting up the Environment
There are many protections in current compilers and operating systems to stop stack attacks like the one we want to do. We have to disable some security options to allow the exploitation to work (Note that the VM image you get has configured the environment).

Disable Address Space Layout Randomization
Address Space Layout Randomization (ASLR) is a security features used in most Operating System today. ASLR randomly arranges the address spaces of processes, including stack, heap, and libraries. It provides a mechanism for making the exploitation hard to success. You can configure ASLR in Linux using the /proc/sys/kernel/randomize_va_space interface. The following values are supported:

0 – No randomization
1 – Conservative randomization
2 – Full randomization

Disable ASLR, run:

$ echo 0 > /proc/sys/kernel/randomize_va_space

Enable ASLR, run:

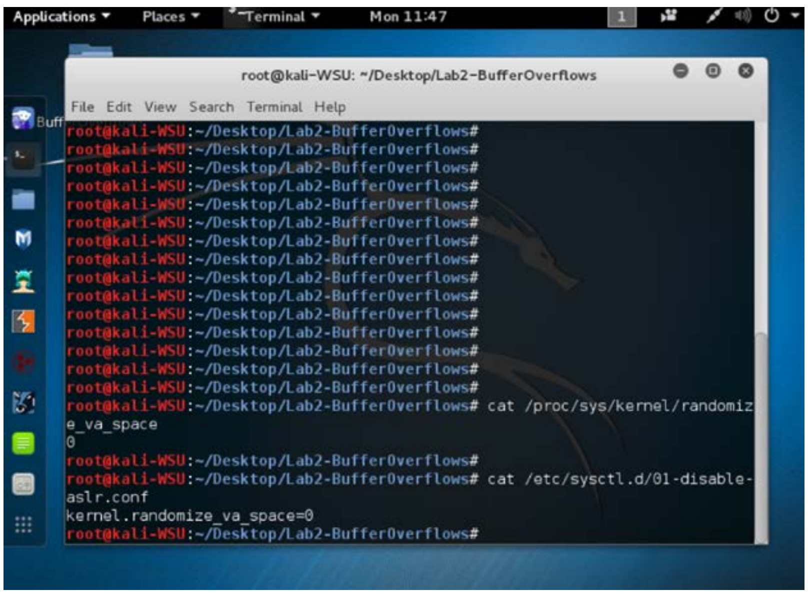
$ echo 2 > /proc/sys/kernel/randomize_va_space

Note that you will need root privilege to configure the interface. Using vi to modify the interface may have errors. The screenshot below shows the value of
/proc/sys/kernel/randomize_va_space

However, this configuration will not survive after a reboot. You will have to configure this in sysctl. Add a file /etc/sysctl.d/01-disable-aslr.conf containing:
kernel.randomize_va_space = 0

This will permanently disable ASLR.

The screenshot below shows you the ASLR configuration. You should open a terminal and try it out.



Set compiler flags to disable security features
When you compile the vulnerable program (explain in the next section) with gcc (C language compiler), use the following compiler flags to disable the security features.

-z execstack

Turn off the NX (no-execute) protection to make the stack executable

-fno-stack-proector

Remove StackGuard that detects stack smashing exploitations

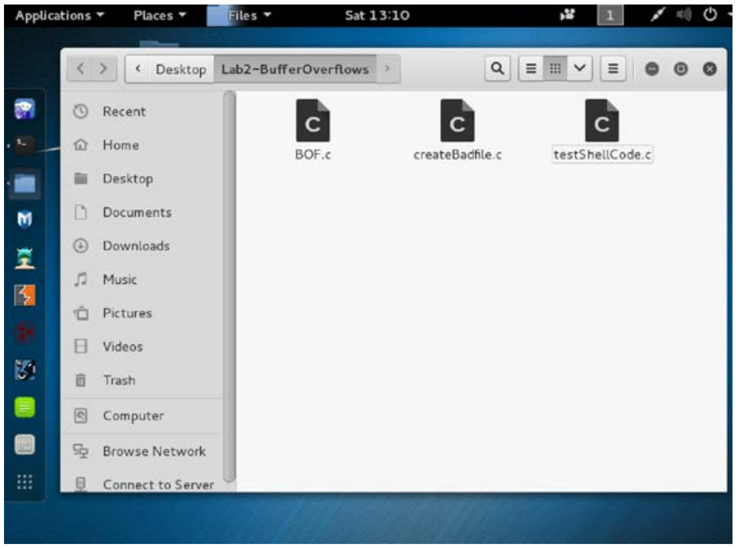
-g

Enable the debugging symbols

Overview

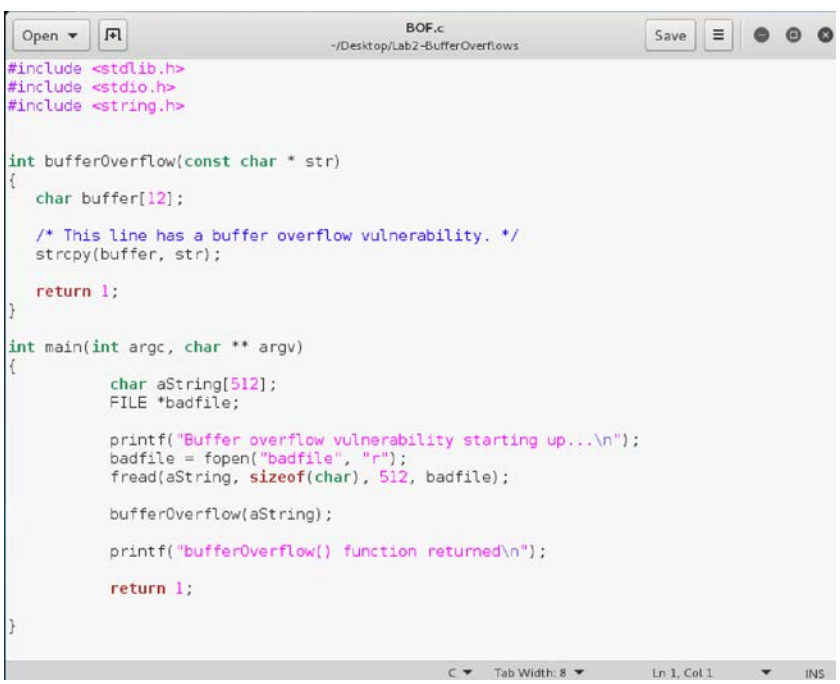
The goal of the exploitation is to teach you how buffer overflows work. **You must gain a shell by passing a malicious input into a vulnerable program.** The vulnerability takes as input a file named "badfile". Your job is to create a badfile that results in the vulnerable program producing a shell. Note that you also have a nop sled to make the vulnerability work even if your shellcode moves by a few bytes. In the Lab2-BufferOverflows folder, it contains the C files you need to use. The screenshot below shows that.

(nop sled: https://www.coengoedegebure.com/buffer-overflow-attacks-explained/ (search for “nop sled” to locate the section “NOP-sled”)



BOF.c

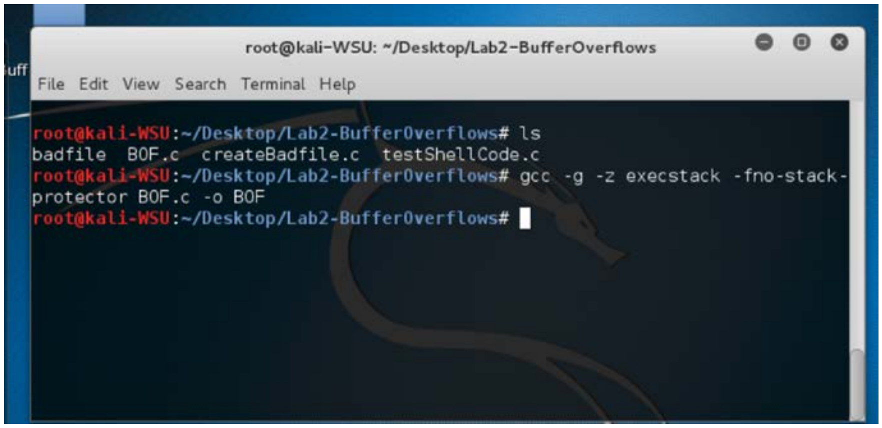
In BOF.c there is an un-bounded strcpy, which means anything that is not null-terminated will overwrite the buffer boundaries and (hopefully) put some information into the stack that you will design. Your exploit must work with the provided version of BOF.c (can't change it to make your code work).




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

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

   /* This line has a buffer overflow vulnerability. */
   strcpy(buffer, str);

   return 1;
}

int main(int argc, char ** argv)
{
           char aString[512];
           FILE *badfile;

           printf("Buffer overflow vulnerability starting up...\n");
           badfile = fopen("badfile", "r");
           fread(aString, sizeof(char), 512, badfile);

           bufferOverflow(aString);

           printf("bufferOverflow() function returned\n");

           return 1;

}
```

To compile BOF.c, you need to add the compile flags mentioned.

$ gcc –g –z execstack –fno-stack-protector BOF.c –o BOF



testShellCode.c
This program simply lets you test shell code itself. There are a lot of different "shell codes" you can find or create, and this is a good way to see what they do, and whether they'll work for you (on your operating system).

The actual shellcode you are using is simply the assembly version of this C code:

```
#include <stdio.h>
int main( ) {
    char *name[2];
    name[0] = "/bin/sh";
    name[1] = NULL;
    execve(name[0], name, NULL);
}
```

int execve(const char *pathname, char *const argv[], char *const envp[]);

execve() executes the program referred to by pathname. This causes the program that is currently being run by the calling process to be replaced with a new program, with newly initialized stack, heap, and (initialized and uninitialized) data segments. pathname must be either a binary executable, or a script.

argv is an array of pointers to strings passed to the new program as its command-line arguments. By convention, the first of these strings (i.e., argv[0]) should contain the filename associated with the file being executed. The argv array must be terminated by a NULL pointer. (Thus, in the new program, argv[argc] will be NULL.)

envp is an array of pointers to strings, conventionally of the form key=value, which are passed as the environment of the new program. The envp array must be terminated by a NULL pointer.

(reference: https://man7.org/linux/man-pages/man2/execve.2.html)

testShellCode.c
~/Desktop/Lab2-BufferOverflows

```c
/*
A program that creates a file containing code for launching shell
*/
#include <stdlib.h>
#include <stdio.h>

//const char code[] = "\xeb\x19\x31\xc0\x31\xdb\x31\xd2\x31\xc9\xb0\x04\xb3\x01\x59\xb2\x05\xcd"\
        "\x80\x31\xc0\xb0\x01\x31\xdb\xcd\x80\xe8\xe2\xff\xff\xff\x68\x65\x6c\x6c\x6f"; // Say Hello


const char code[] = \
"\x31\xc0"          /* Line 1:  xorl    %eax,%eax       */      \
"\x50"              /* Line 2:  pushl   %eax            */      \
"\x68""//sh"        /* Line 3:  pushl   $0x68732f2f     */      \
"\x68""/bin"        /* Line 4:  pushl   $0x6e69622f     */      \
"\x89\xe3"          /* Line 5:  movl    %esp,%ebx       */      \
"\x50"              /* Line 6:  pushl   %eax            */      \
"\x53"              /* Line 7:  pushl   %ebx            */      \
"\x89\xe1"          /* Line 8:  movl    %esp,%ecx       */      \
"\x99"              /* Line 9:  cdql                    */      \
"\xb0\x0b"          /* Line 10: movb    $0x0b,%al       */      \
"\xcd\x80"          /* Line 11: int     $0x80           */      \
;

int main(int argc, char ** argv)
{
  int (*func)();
  func = (int (*)()) code;
  (int)(*func)();

  return 0;
}
```

C ▾ Tab Width: 8 ▾ Ln 17, Col 66 ▾ INS

createBadfile.c

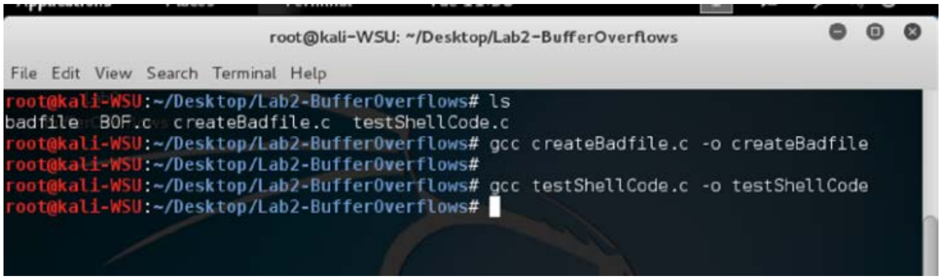
This program writes out "badfile", however currently it is just full of nops (no ops). You need to modify it to place your shell code into it and cause the code to jump to the shellcode. The shellcode included already in badfile (as a char array) does work. You shouldn't need to modify it, but you're welcome to.

createBadfile.c
~/Desktop/Lab2-BufferOverflows

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

//const char shellcode[] = "\xeb\x19\x31\xc0\x31\xdb\x31\xd2\x31\xc9\xb0\x04\xb3\x01\x59\xb2\x05\xcd"\
        "\x80\x31\xc0\xb0\x01\x31\xdb\xcd\x80\xe8\xe2\xff\xff\xff\x68\x65\x6c\x6c\x6f"; // Say Hello

const char shellcode[] = \
"\x31\xc0"          /* Line 1:  xorl    %eax,%eax       */      \
"\x50"              /* Line 2:  pushl   %eax            */      \
"\x68""//sh"        /* Line 3:  pushl   $0x68732f2f     */      \
"\x68""/bin"        /* Line 4:  pushl   $0x6e69622f     */      \
"\x89\xe3"          /* Line 5:  movl    %esp,%ebx       */      \
"\x50"              /* Line 6:  pushl   %eax            */      \
"\x53"              /* Line 7:  pushl   %ebx            */      \
"\x89\xe1"          /* Line 8:  movl    %esp,%ecx       */      \
"\x99"              /* Line 9:  cdql                    */      \
"\xb0\x0b"          /* Line 10: movb    $0x0b,%al       */      \
"\xcd\x80"          /* Line 11: int     $0x80           */      \
;


int main(int argc, char ** argv) {
        char buffer[512];

        FILE *badfile;

        /* Init the buffer with nop (0x90) */
        memset(&buffer, 0x90, 512);


        /* Save to badfile. */
        badfile = fopen("badfile", "w+");
        fwrite(buffer, 512, 1, badfile);
        fclose(badfile);

        printf("Completed writing\n");

        return 0;
}
```

C ▾ Tab Width: 8 ▾ Ln 1, Col 1 ▾ INS

To compile the testShellCode.c and createBadfile.c, you do not need to add the compile flags mentioned early. You can just simply compile it with gcc



Starting the Exploitation

There are really two challenges in the lab. To execute the shellcode, you want to overwrite the return address in the bufferOverflow() function. You must make the return address of that function point to your shellcode.

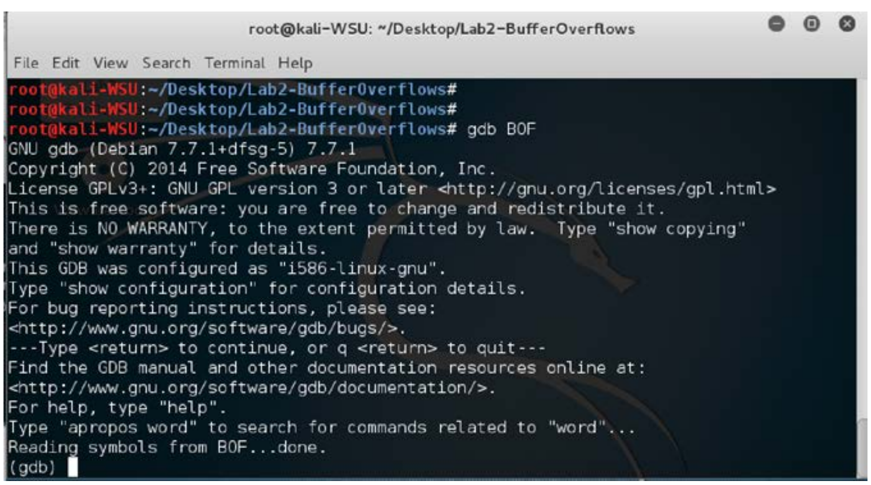
1. You need to figure out what memory address the return address is stored in.
2. Then you need to figure out the address of your shellcode in memory, and write the shellcode's address into the return address you found in step 1.

In the lab instruction, I will give you some hints for the step 1.

Finding Return Address on the Stack

In order to find the return address on stacks, we first use GDB, The GNU Project Debugger, to take a look at the assembly code. You can find more information about GDB from here: https://www.gnu.org/software/gdb/. Note that you can also use tool, objdump (https://man7.org/linux/man-pages/man1/objdump.1.html, https://www.thegeekstuff.com/2012/09/objdump-examples/), to read the assembly code.

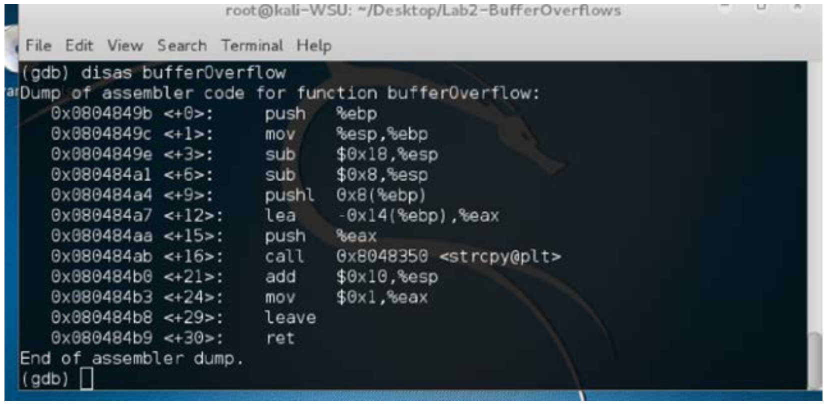
$ gdb BOF



First, we disassemble the main() function of the BOF program. We find the bufferOverflow() function in the main() function (type disas main in the GDB). Then, we disassemble the bufferOverflow() function, which has a vulnerability in it.

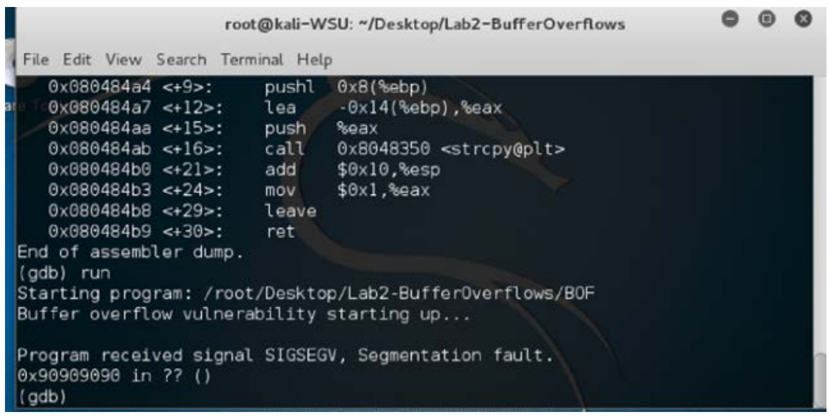
$ (gdb) disas main

$ (gdb) disas bufferOverflow



You need to understand the assembly code to find where the return address is on the stack. Next, type run in the GDB to execute the BOF program.

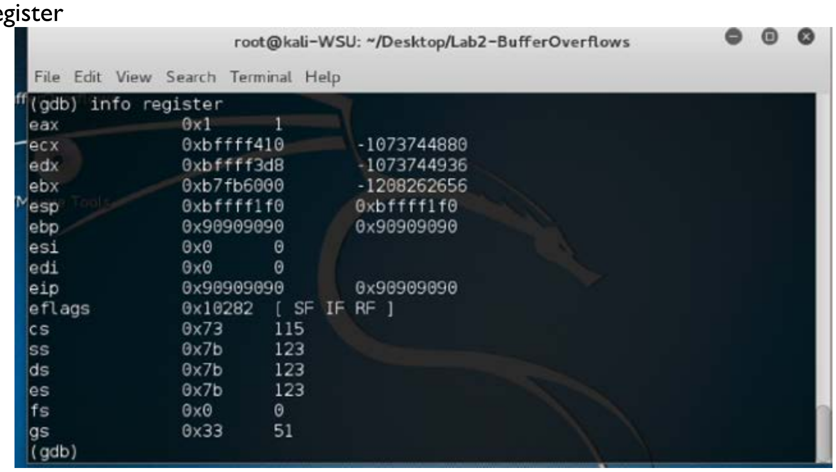
$ (gdb) run



As we expected, the BOF program generates an exception, segmentation fault. The Instruction Pointer = (EIP) is 0x90909090. This is because we put NOP sleds on the badfile that overflows the buffer in the BOF program.

You also can see more register information by execute info register in the GDB

$ (gdb) info register



Note that you can always type help in the GDB to learn the commands.

Questions for the Lab

Software Requirements
All required tools are packed in the provided Lab 1 virtual machine.

- VMWare Software
  - o https://www.vmware.com/
- VirtualBox Software
  - o https://www.virtualbox.org/

The Lab 2 virtual machine can be downloaded from https://drive.google.com/file/d/1IxkcpB4C11mXYVygo4IKpcieXJfrx6XB/view?usp=sharing
Login the Kali Linux with the username [root] without the brackets, and the passcode [TBA in the class] without the brackets.

Your submission should be a zip file containing:
1. Your updated createBadfile.c that generates the input for the BOF program. [2.5 pts]
2. A copy of the badfile. This must generate a shell when BOF runs from the command line in the VM. [2.5 pts]
3. A screenshot of using BOF program to gain a shell (see simple screenshot below). [1 pt]
4. A WORD document with answers to the following questions:
   a. What happens when you compile without "-z execstack"? [1 pt]
   b. What happens if you enable ASLR? [1 pt] Does the return address change? [1 pt]
   c. Does the address of the buffer[] in memory change when you run BOF using GDB, /home/root/Desktop/Lab2-BufferOverflows/BOF, and ./BOF? [1 pt]

Happy Exploiting!