Stack and Stack Frame

CSC 472 - 01

Tyler Prehl

9/20/2021

Introduction

The purpose of this lab is to gain a stronger understanding of the stack and memory. This will be achieved through the analysis of a simple program’s assembly language and the tracking of the registers, stack, and memory. With a stronger understanding of the stack and memory, I will be able to properly understand the concepts behind “stack overflow” security attacks.

Analysis and Results

“lab1.c” is a simple program that is shown below. In this program, two integers are added together with an additional 1 using the “add\_plus1” method, and the value is printed to the terminal. Although the references to the memory address where the actual values are stored are passed into the method through the parameters, the addresses to the actual values are stored once more in variables x and y. The returned value is then x+y+1, which is printed to the terminal in the main function, and the program comes to completion.

Text, letter

Description automatically generated

To analyze this program further to track the registers, stack, and memory, we must disassemble the code using the gdb debugger. Once disassembled, we get the following output:

Graphical user interface, text

Description automatically generated

The fourth and fifth lines:



represent the ebp (stack *base* pointer) being pushed to the top of the stack to create the bottom of the stack frame, and the value stored in ebp being copied into esp (current stack pointer) so that esp once again references the top of the stack. In addition, we can see that the assembler code used for creating the stack frame of the main() function is those same fourth and fifth lines of assembly instructions.

You can also see on the following lines that the values 5 and 6 are pushed on the stack at 14 and 10 bytes “above” the ebp register respectively.



Furthermore, we can see from stepping through the code that before the function add\_plus1 call, there are actually two sets of 5 and 6 saved onto the stack:

A screenshot of a computer

Description automatically generated with medium confidence

This is because the values 5 and 6 are initially stored when they are instantiated into the variables (memory locations) a and b, and then are stored once again in different memory locations to be passed into the function call add\_plus1. In fact, they actually get stored on the stack a third time when the references to 5 and 6 are saved into the x and y variables in the add\_plus1 function:

Graphical user interface, text

Description automatically generated

As you can see from the stack address, these values 5 and 6 are further up the stack than the other two sets of 5 and 6, allowing the user to discern between the first two sets and this set. We can also disassemble the function add\_plus1 to receive the following output:

Graphical user interface, text

Description automatically generated

Once again, we see the “push ebp” and “mov ebp, esp” assembly instructions which signify and create the start of the new stack frame. The last thing of note in the disassembled assembly code is that the final product (the 5+6+1 value) is stored in the register eax:

Graphical user interface, text, application

Description automatically generated

where the stored values in eax and edx at the time of the first line are 6 and 5, as expected for the 1 to be added in directly in the following line.

Graphical user interface, text

Description automatically generated

In the code:x86:32 section of the gdb debugger, it looks like this:



Discussion and Conclusion

Throughout this lab, I utilized the gdb debugger to analyze the stack, memory, registers, and disassembled code for the executable code created by the compiling of lab1.c. After practicing navigating gdb and using it to understand all the “behind the scenes” work of moving and interacting with data within memory, I feel better prepared to understand the concepts behind stack overflow attacks since these attacks revolve around overflowing the stack with extra data to lead to a different place in memory where malicious code can be executed.