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.

```
#include <stdio.h>
int add_plus1(int a, int b)
{
    int x = a;
    int y = b;
    return x+y+1;
}

int main(int argc, char** argv) {
    int a=5, b=6;
    int d = add_plus1(a,b);
    printf("%d\n", d);
    return 0;
}
```

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:

```
gef➤  disas main
Dump of assembler code for function main:
   0x0804919f <+0>:     lea    ecx,[esp+0x4]
   0x080491a3 <+4>:     and    esp,0xfffffff0
   0x080491a6 <+7>:     push   DWORD PTR [ecx-0x4]
   0x080491a9 <+10>:    push   ebp
   0x080491aa <+11>:    mov    ebp,esp
   0x080491ac <+13>:    push   ebx
   0x080491ad <+14>:    push   ecx
   0x080491ae <+15>:    sub    esp,0x10
   0x080491b1 <+18>:    call   0x80490b0 <__x86.get_pc_thunk.bx>
   0x080491b6 <+23>:    add    ebx,0x2e4a
   0x080491bc <+29>:    mov    DWORD PTR [ebp-0x14],0x5
   0x080491c3 <+36>:    mov    DWORD PTR [ebp-0x10],0x6
   0x080491ca <+43>:    push   DWORD PTR [ebp-0x10]
   0x080491cd <+46>:    push   DWORD PTR [ebp-0x14]
   0x080491d0 <+49>:    call   0x8049176 <add_plus1>
   0x080491d5 <+54>:    add    esp,0x8
   0x080491d8 <+57>:    mov    DWORD PTR [ebp-0xc],eax
   0x080491db <+60>:    sub    esp,0x8
   0x080491de <+63>:    push   DWORD PTR [ebp-0xc]
   0x080491e1 <+66>:    lea    eax,[ebx-0x1ff8]
   0x080491e7 <+72>:    push   eax
   0x080491e8 <+73>:    call   0x8049040 <printf@plt>
   0x080491ed <+78>:    add    esp,0x10
   0x080491f0 <+81>:    mov    eax,0x0
   0x080491f5 <+86>:    lea    esp,[ebp-0x8]
   0x080491f8 <+89>:    pop    ecx
   0x080491f9 <+90>:    pop    ebx
   0x080491fa <+91>:    pop    ebp
   0x080491fb <+92>:    lea    esp,[ecx-0x4]
   0x080491fe <+95>:    ret
End of assembler dump.
gef➤
```

The fourth and fifth lines:

```
 0x080491a9 <+10>:    push   ebp
 0x080491aa <+11>:    mov    ebp,esp
```

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.

```
0x080491bc <+29>:    mov    DWORD PTR [ebp-0x14],0x5
0x080491c3 <+36>:    mov    DWORD PTR [ebp-0x10],0x6
```

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:

```
0xffffd678│+0x0000: 0x00000005    ← $esp
0xffffd67c│+0x0004: 0x00000006
0xffffd680│+0x0008: 0x00000002
0xffffd684│+0x000c: 0x00000005
0xffffd688│+0x0010: 0x00000006
0xffffd68c│+0x0014: 0x08049231  →  <__libc_csu_init+33> lea ebx,
0xffffd690│+0x0018: 0xffffd6b0  →  0x00000002
0xffffd694│+0x001c: 0x00000000

    0x80491c3 <main+36>        mov    DWORD PTR [ebp-0x10], 0x6
    0x80491ca <main+43>        push   DWORD PTR [ebp-0x10]
    0x80491cd <main+46>        push   DWORD PTR [ebp-0x14]
```

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:

```
0xffffd660│+0x0000: 0xf7fb5000  →  0x001e4d6c    ← $esp
0xffffd664│+0x0004: 0xf7fe4080  →   push ebp
0xffffd668│+0x0008: 0x00000005
0xffffd66c│+0x000c: 0x00000006
0xffffd670│+0x0010: 0xffffd698  →  0x00000000    ← $ebp
```

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:

```
gef➤  disas add_plus1
Dump of assembler code for function add_plus1:
   0x08049176 <+0>:     push   ebp
   0x08049177 <+1>:     mov    ebp,esp
   0x08049179 <+3>:     sub    esp,0x10
   0x0804917c <+6>:     call   0x80491ff <__x86.get_pc_thunk.ax>
   0x08049181 <+11>:    add    eax,0x2e7f
   0x08049186 <+16>:    mov    eax,DWORD PTR [ebp+0x8]
   0x08049189 <+19>:    mov    DWORD PTR [ebp-0x8],eax
   0x0804918c <+22>:    mov    eax,DWORD PTR [ebp+0xc]
   0x0804918f <+25>:    mov    DWORD PTR [ebp-0x4],eax
   0x08049192 <+28>:    mov    edx,DWORD PTR [ebp-0x8]
   0x08049195 <+31>:    mov    eax,DWORD PTR [ebp-0x4]
=> 0x08049198 <+34>:    add    eax,edx
   0x0804919a <+36>:    add    eax,0x1
   0x0804919d <+39>:    leave
   0x0804919e <+40>:    ret
End of assembler dump.
gef➤
```

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:

```
  0x08049192 <+28>:    mov    edx,DWORD PTR [ebp-0x8]
  0x08049195 <+31>:    mov    eax,DWORD PTR [ebp-0x4]
> 0x08049198 <+34>:    add    eax,edx
  0x0804919a <+36>:    add    eax,0x1
```

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.

```
$eax   : 0x6
$ebx   : 0x08
$ecx   : 0xff
$edx   : 0x5
```

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

```
→ 0x8049198 <add_plus1+34>   add    eax, edx
  0x804919a <add_plus1+36>   add    eax, 0x1
```

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.