West Chester University

CSC 472

Dr. Si Chen

Fall 2023 Lab 1

Submitted by

Sean Berlin

9/20/23

1. **Introduction:** The purpose of this lab was for students to gain a deeper understanding of stack frame manipulation and assembly code by reverse engineering a program written in C. To be specific, they utilized GDB to analyze the binary program in order to reverse engineer and dissect the assembly code.

2. **Analysis and Results:**

Questions:

Q1: Identify the assembly instructions for creating the stack frame of the main() function. (1 point)

```
0x080496e8 <+4>:     and    esp,0xfffffff0
0x080496eb <+7>:     push   DWORD PTR [ecx-0x4]
0x080496ee <+10>:    push   ebp
0x080496ef <+11>:    mov    ebp,esp
0x080496f1 <+13>:    push   ecx
0x080496f2 <+14>:    sub    esp,0x14
```

Figure 1

These assembly instructions are responsible for creating the stack frame of the main() function.

```
0x080496ef <+11>:    mov    ebp,esp
```

Figure 2

This instruction sets up the base pointer, ebp, to point to the current stack frame effectively establishing the stack frame for the main() function.

Q2: Identify and explain the purpose of the two lines related to setting variables p and q. (1 point)

```
0x08049703 <+31>:    push   DWORD PTR [ebp-0x10]
0x08049706 <+34>:    push   DWORD PTR [ebp-0xc]
```

Figure 3

push DWORD PTR [ebp-0x10]: This instruction pushes the value of the variable q onto the stack. It stores the variable q at the memory location [ebp-0x10].
push DWORD PTR [ebp-0xc]: This instruction pushes the value of the variable p onto the stack. It stores the variable p at the memory location [ebp-0xc].
In summary, these two instructions prepare the values p and q to be passed as arguments when calling the multiply by two() function.

Q3: Before calling multiply by two(), why does the stack contain two sets of “3,4” instead of just one set (see Figure.1)? (1 point)
Figure 1: Two sets of “3,4”

The stack contains two sets of "3,4" before calling the multiply_by_two() function because in the C language, it requires arguments to be pushed onto the stack in reverse order. This is done to ensure that the function receives the correct arguments when it is called. The stack is responsible for managing these arguments and function calls.

Q4: Explain the meaning of add eax,edx and add eax,eax. Why not using mul (Multiply) instruction instead (take a guess)? (1 point)
add eax, edx: This instruction adds the value from the edx register to the value in the eax register and ultimately stores the result into eax.
add eax, eax: This instruction adds the value from the eax register to itself and stores it there.
A best guess for why the mul (multiply) instruction is not used is due to optimization. Addition is often faster than multiplication in terms of execution time. Using the add instruction twice to double a value may be more efficient than using mul for something as simple as multiplying by two. To summarize, ultimately for speed and simplicity advantages.

Q5: Which register is used to store the final multiplication result? (1 point)

Figure 4

As shown in the above screenshot, the eax register is used to accumulate the result of the additions and multiplications. It then holds the final result when the function returns.

Analysis:
By using the GDB debugger, the exploration of stack frame manipulation and assembly code was achieved. *Figure 1* shows the core concept of how the stack organizes itself during function calls with stack frames, base pointers, and stack pointers. In addition, *figure 3* revealed that function arguments were pushed onto the stack in reverse order, thus following the C language calling conventions for parameter passing. Furthermore, the use of add instructions for multiplication by 2 was observed. The student hypothesized this was due to efficiency measures. These findings highlighted the crucial role of the stack in function execution and provided further insight into practical assembly programming techniques.

3. Discussion and Conclusion

Generated by ChatGPT- Upon reflection, the lab effectively fulfilled its stated purpose of enhancing the student's comprehension of stack frame manipulation and assembly code analysis. The student successfully employed GDB to reverse engineer and dissect the provided C program, gaining practical insights into stack organization during function calls and returns. Notably, the observed practice of pushing function arguments in reverse order aligned with theoretical expectations rooted in C calling conventions. Moreover, the preference for *add* instructions over the *mul* instruction for simple multiplications highlighted the importance of efficient assembly programming. This hands-on experience solidified the student's understanding of stack manipulation and assembly code principles, paving the way for more adept software security practices in the future.