1. **Introduction / Purpose / Intent**

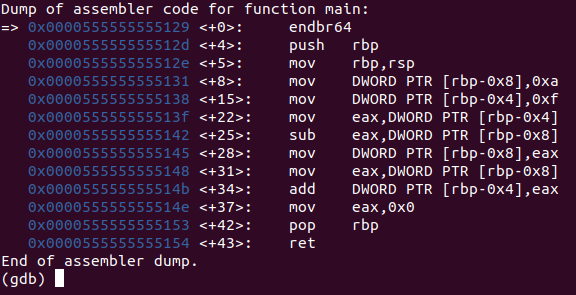
For this lab I was tasked to, using the disassemble capability of the gbd debug utility, reverse engineer 3 provided executable files named Example1, Example2, and Example3. The objective listed is “Practice reverse engineering of executable software”. At the top of the canvas lab page I am instructed to submit 3 “Exercise.c” files along with this lab report. I interpreted this to mean I would be generating a program that would operate similarly to how each of the executable examples operated.

On the lab page on canvas the process to reverse engineer an executable file is provided. This process consists of a number of easy to follow steps and expected output from an example program named “sample.c”. Essentially the order is to run the program in gdb, set a breakpoint at the start of the main() function to pause the program before it executes, run the program and arrive at the breakpoint at main(), disassemble the program using disassemble in gbd. After this process is explained, the output of disassemble from “sample.c” is annotated to explain what each of the steps is preforming. After this, the lab challenges the student to see if we can reverse engineer the executable programs provided, and reminds us execute permission is needed on the files to do so.

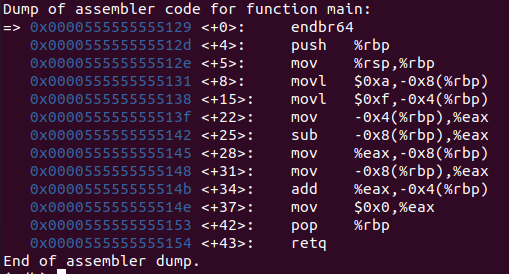
1. **Process**

The first step in this lab was to analyze the “sample.c” code and disassemble output of assembly code. The program itself is very simple, declaring 2 integer variables (X, Y) into existence, assigning each a value (10, 15), and finally performing 2 arithmetic calculations on the variables before ending (x = y-x, y = x+y). I began going line by line through the assembly code, attempting to decipher it and comparing my interpretation to the provided comment annotation associated with it. This process took about 30 minutes before I believed I had a good grasp on the output of the “sample.c” assembly and associated comments. Below are my results from following the instructions provided in the lab page for the “sample.c” program along with the provided annotations that explain what the program is accomplishing.

Sample (Intel)



Sample (AT&T)



PROVIDED ANNOTATION OF ASSEMBLY INSTRUCTION

push %rbp # save the old value of %rbp

mov %rsp,%rbp # set %rbp to the value of %rsp

movl $0xa,-0x8(%rbp) # x = 10

movl $0xf,-0x4(%rbp) # y = 15

mov -0x4(%rbp),%eax # load y into %eax

sub -0x8(%rbp),%eax # y - x is now in %eax

mov %eax,-0x8(%rbp) # x = y - x

mov -0x8(%rbp),%eax # load x into %eax

add %eax,-0x4(%rbp) # y = x + y

mov $0x0,%eax

pop %rbp # restore the old value of %rbp

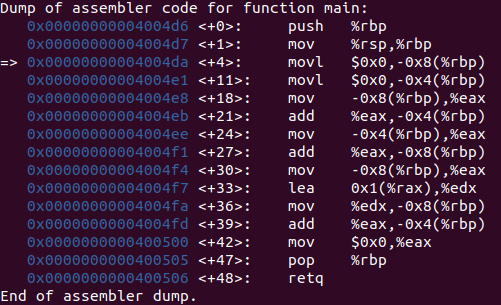
retq # return from main()

Following this exercise I thought the next best step would be to produce screenshots to decipher for each of the 3 example executable files. Below are those screenshots.

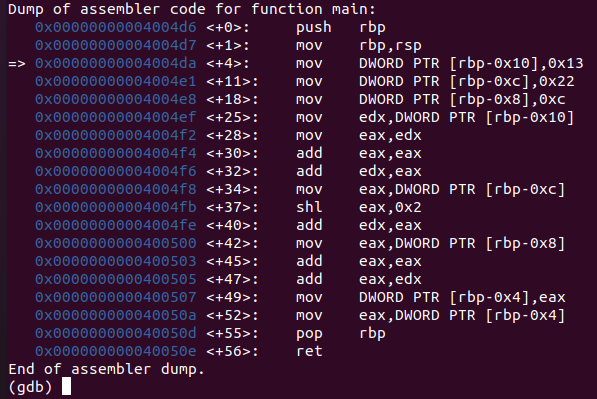
Example1 (Intel)



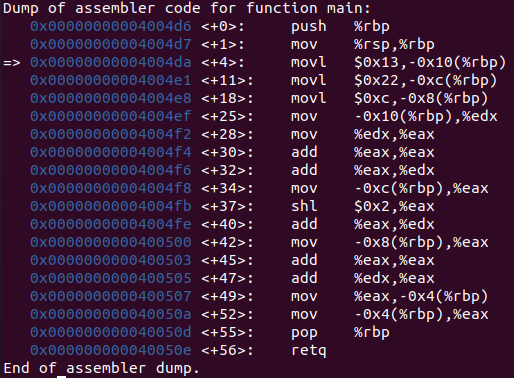
Example1 (AT&T)



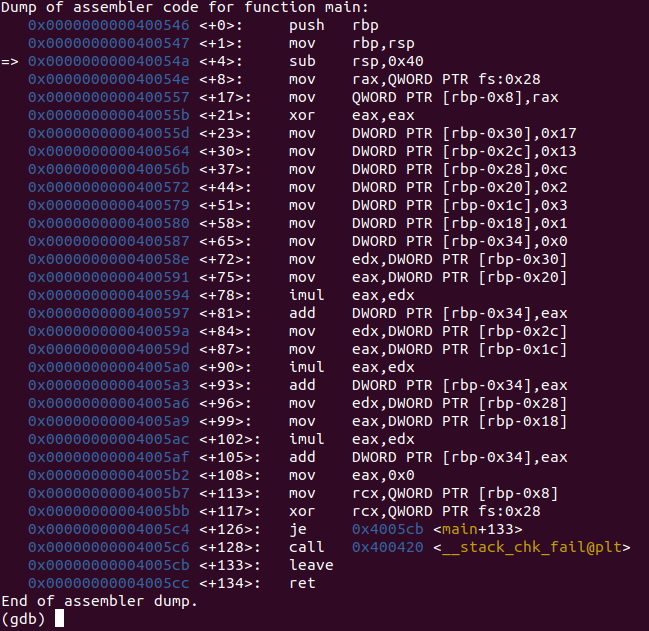
Example2 (Intel)



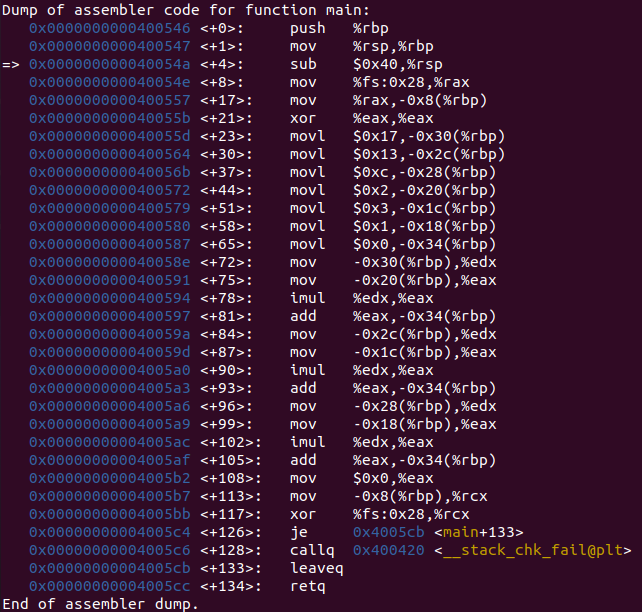
Example2 (AT&T)



Example3 (Intel)



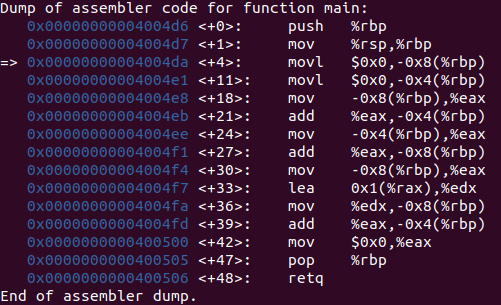
Example 3 (AT&T)



Now that I had the disassemble output assembly code for each of the example executable files, I could repeat the process I initially performed with the “sample.c” file with a slight alteration to the process. I will be generating the annotations of each line into pseudo-code similarly as how it was done for me in the sample file. I assumed Example1 would be the simplest so I began with it.

* ***Example1***

Example1 (AT&T)



push %rbp # save the old value of %rbp

mov %rsp,%rbp # set %rbp to the value of %rsp

movl $0x0,-0x8(%rbp) # x = 0

movl $0x0,-0x4(%rbp) # y = 0

mov -0x8(%rbp),%eax # load x into %eax

add %eax,-0x4(%rbp) # x = x+y

mov -0x4(%rbp),%eax # load y into %eax

add %eax,-0x8(%rbp) # y = y+x

mov -0x8(%rbp),%eax # load x into %eax

lea 0x1(%rax),%edx # stores %edx into register 1

mov %edx,-0x8(%rbp) # load x into %edx

add %eax,-0x4(%rbp) # y = y+x

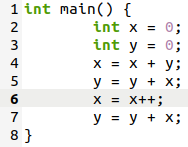
mov $0x0,%eax #

pop %rbp # restore the old value of %rbp

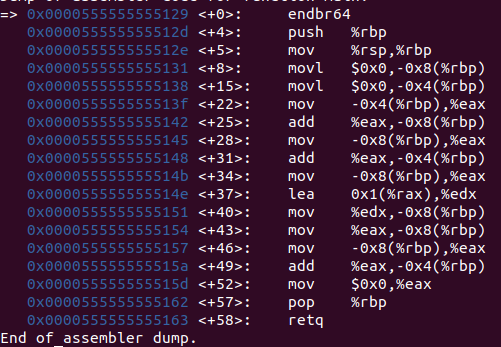
retq # return from main()

After I generated some pseudo-code I returned to my virtual machine to generate some code that would produce this disassemble. Below is the resulting program and disassemble output. Highlighted in between the parallel light blue lines is a section where the assembly code indicated that I had had the variables being assigned on lines 4-7 incorrect, so I revised the code and reran the disassemble, which seemed to produce the desired result. Highlighted in the blue box is an additional line of code that I cannot determine its origin.

Exercise1.c

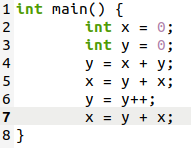


Exercise1 gdb disassemble

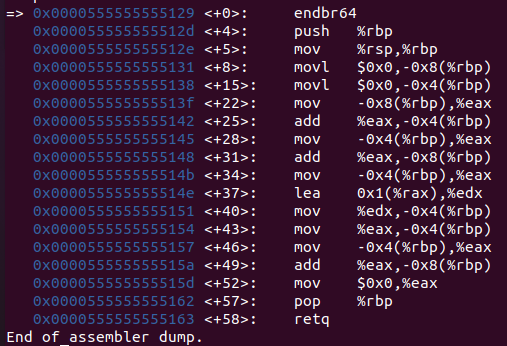


Exercise 1 [Left] and Example1 (AT&T) [Right]

Corrected Exercise1.c

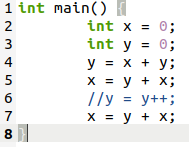


Corrected Exercise1 assembly code

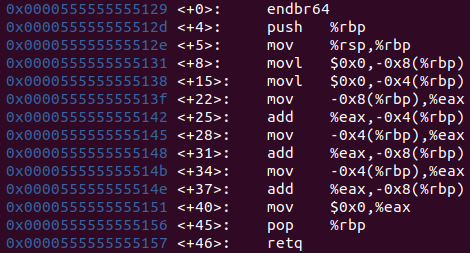


I attempted to remove the line of code containing the increment operator for the y variable to see if it would remove the additional 2 lines of assembly language, however it only removed the “lea” statement on line <+37> of assembly, as seen below, and so have left it as is.

Exercise1.c (Line 6 commented out)

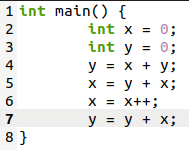


Exercise1 assembly code (Line 6 commented out)



I did discover that my increment statement had been performed on the wrong variable, and have changed the code to reflect this as can be seen below. In addition, the assignment for line 7 was incorrect, and has been changed. I believe the Exercise1.c file is, to the best of my ability, as close as I can get to the Example1 executable. Below is the final revision of the Excersise1.c file along with the assembly output by gdb. The red box indicates the lines from my code that are not present in the code of the Excersize1 executable. The parallel blue lines have been maintained for reference of the first draft.

Exercise1.c

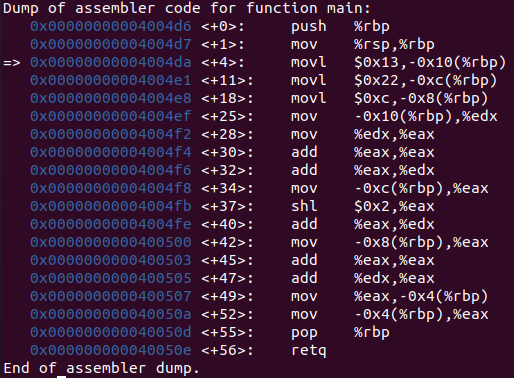


Exercise1 [Left] and Example1 [Right]



* ***Example2***

Example2 (AT&T)



push %rbp # save the old value of %rbp

mov %rsp,%rbp # set %rbp to the value of %rsp

movl $0x13,-0x10(%rbp) # x = 0x13 = 19

movl $0x22,-0xc(%rbp) # y = 0x22 = 34

movl $0xc,-0x8(%rbp) # z = 0xc = 12

mov -0x10(%rbp),%edx # load x into %edx

mov %edx,%eax # load %edx into %eax

add %eax,%eax # %eax = %eax + %eax

add %eax,%edx # %edx = %eax + %eax

mov -0xc(%rbp),%eax # load y into %eax

shl $0x2,%eax # shift %eax left 2

add %eax,%edx # %edx = %eax + %eax

mov -0x8(%rbp) ,%eax # load z into %eax

add %eax,%eax # %eax = %eax + %eax

add %edx,%eax # %eax = %edx + %eax

mov %eax,-0x4(%rbp) # load %eax into pointer-4

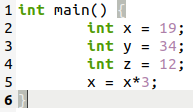
mov -0x4(%rbp),%eax # load pointer-4 into %eax

pop %rbp # restore the old value of %rbp

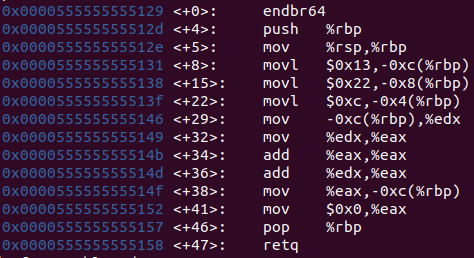
retq # return from main()

To start with Example2, after I had generated the pseudo-code, I declared into existence 3 variables and assigned them the values indicated by the assembly. Then it seemed to be that the x variable would be loaded into the %eax and added to itself, then the original value would be added to it a third time. To me this sounded like a multiplication by 3, so I entered it into the program, compiled it, and tested it as can be seen below.

Exercise2.c

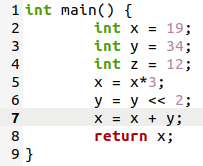


Exercise2 assembly



I had successfully completed the first 9 lines of pseudo-code; however I did hold one small issue. The variables I had declared were exactly 4 bytes off of their intended values. This indicated to me that there would be another integer variable somewhere in the program. I checked the Example2 assembly and located at <+49> the value of %eax gets stored into the missing variable. Keeping this note I continued through the lines of code to operate on the y variable. This value would get shifted left 2 and then added to the value of %eax that I had operated on earlier. I added these lines to the program, compiled and checked the assembly as can be seen below.

Excersize2.c

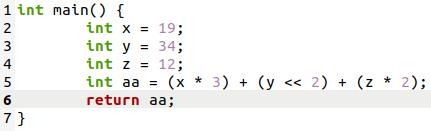


Excersize2 assembly



As can be seen the operations performed are being performed on the variable location, and not on %eax and %edx as is in the Example2 assembly code. I queried my fellow students as to what might cause this and one colleague made me consider that I was working too slowly. For example instead of accomplishing what I had in 3 lines, perhaps I needed to accomplish it all in a single line to maintain the %eax and %edx values. I attempted this in my next rendition of the code as is shown below. I also implemented the 4th variable as the variable to hold the result of the 1 line operation, as well as the code to operate on the Z variable which simply gets added to itself and then to the value of %eax.

Exercise2.c



Exercise2 assembly code



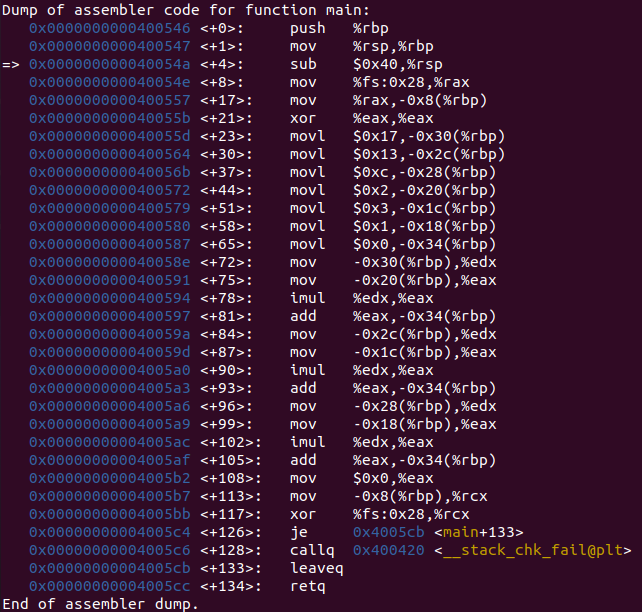
This result not only cleared up my issue with my variables being stored in the wrong place on the stack, but also produced an exact match between the Example2 and Exercise2 assembly which can be seen below in-between the parallel blue lines.

Exercise2 [Left] and Example2 [Right] Assembly code



* ***Example3***

Example3 (AT&T)



push %rbp # save the old value of %rbp

mov %rsp,%rbp # set %rbp to the value of %rsp

sub $0x40,%rsp # Subtract 64 from the stack pointer

mov %fs:0x28,%rax #

mov %rax,-0x8(%rpb) #

xor %eax,%eax # Zero out %eax

movl $0x17,-0x30(%rpb) # x[0] = 0x17 = 23

movl $0x13,-0x2c(%rpb) # x[1] = 0x13 = 19

movl $0xc,-0x28(%rpb) # x[2] = 0xc = 12

movl $0x2,-0x20(%rpb) # x[3] = 0x2 = 2

movl $0x3,-0x1c(%rpb) # x[4] = 0x3 = 3

movl $0x1,-0x18(%rpb) # x[5] = 0x1 = 1

movl $0x0,-0x34(%rpb) # y = 0

mov -0x30(%rpb),%edx # load x[0] into %edx

mov -0x20(%rpb),%eax # load x[3] into %eax

imul %edx,%eax # %eax = %edx \* %eax

add %eax,-0x34(%rpb) # y = y + %eax

mov -0x2c(%rpb),%edx # load x[1] into %edx

mov -0x1c(%rpb),%eax # load x[4] into %eax

imul %edx,%eax # %eax = %edx \* %eax

add $eax,-0x34(%rpb) # y = y + %eax

mov -0x28(%rpb),%edx # load x[2] into %edx

mov -0x18(%rpb),%eax # load x[5] into %eax

imul %edx,%eax # %eax = %edx \* %eax

add %eax,-0x34(%rpb) # y = y + %eax

mov %0x0,%eax # load 0 into %eax

mov -0x8(%rbp),%rcx #

xor %fs:0x28,%rcx #

je 0x4005cb<main+133> #

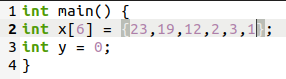
callq 0x400420<\_\_stack\_chk\_fail@plt> #

leaveq #

retq # return from main()

During the lab session on 12 Nov 2020 it was indicated that there would be array business inside Example3, and it makes sense when the assembly code is analyzed. 6 variables are declared at the same time and assigned values, this could be done with a comma separated list of integers, followed by a single instance of an integer being assigned a 0 value, however that doesn’t make logical sense. It does however logically make sense to declare and assign values to an array, and then declare and assign value to the last remaining integer required. I went ahead and placed this into my Exercise3.c program, compiled and checked the assembly as can be seen below.

Exercise3.c

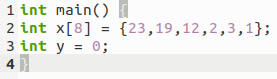


Exercise3 assembly

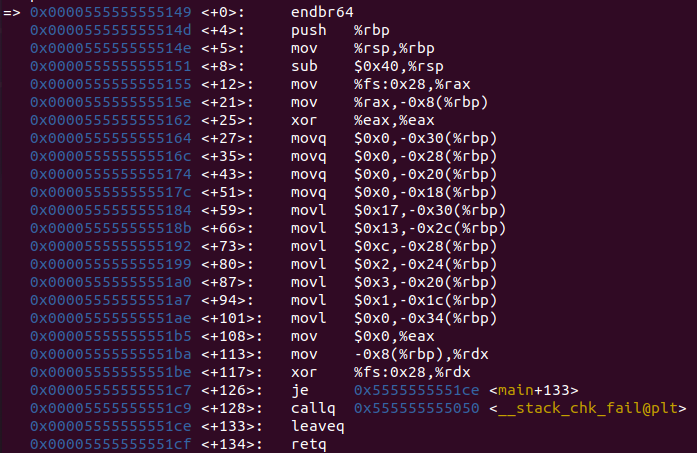


The assembly from exercise1 at this point nearly matches the first 13 lines of Example3 with the exception of line <+4>: sub 0x40 in Example 3 and <+8>: sub 0x30 in Exercise3. I prompted a fellow student for why this may happen and they suggested an overprovisioned array, meaning being declared for more array locations than we have values. I took a closer look at the values and decided to try adding 2 additional spots to the array, declaring into existence x[8] rather than x[6] and successfully matched the Example3 assembly code as can be seen below. Unfortunately, the single line matching made others not, such as the positions within the array, and added additional assignment statements that were not intended to be present. This clearly is not a match, and I reverted to my prior version of code.

Exercise3.c

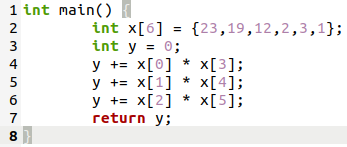


Exercise3 assembly

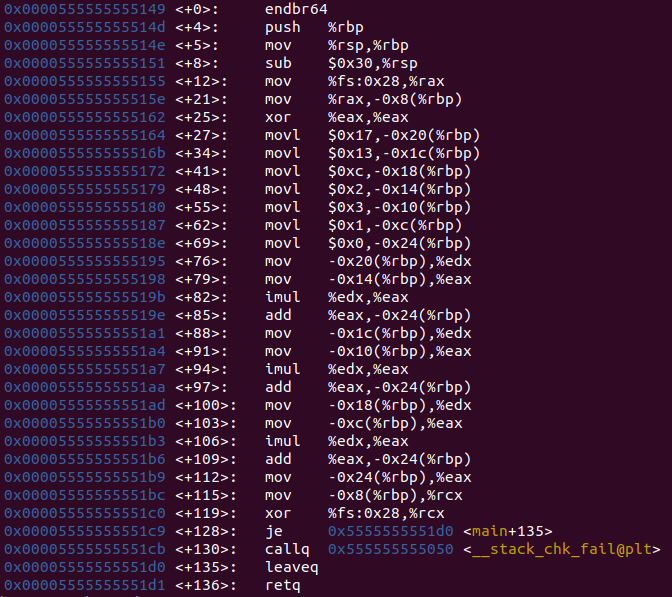


What this did tell me is that I may need 2 additional variables elsewhere in the program, so I scanned through the original Example3 assembly code looking for my 2 potential missing variables. I didn’t find anything that jumped out immediately to me, but I still had tasks I could complete so I moved on to that. The next few steps seemed realitively simple, x[0] and x[3] would be multiplied together and placed into y, then the same would happen with 2 more pairs; x[1] and x[4] as well as x[2] and x[5]. I placed this code into my program, compiled, and checked the assembly code against the assembly from Example3 as can be seen below.

Exercise3.c



Exercise3 assembly



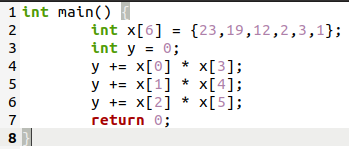
Nearly all of the assembly lines from my program, Exercise3, matched the assembly code from the provided Example3 with a few minor deviations. As discussed my program uses a different value for the sub operation at the beginning of the program, leading to variable value differences on the stack. This has been highlighted inside the first red box in the image below, while the variable values rest between the 2 parallel blue lines. All of the operations match their intended value, and I believe this aligns with what the design of Example3 computes. There is one final deviation in the 2 programs, as towards the end of the program Example3 stores the value 0 into the %eax and my Exercise3 stores the value of the y variable as is highlighted in the second red box below.

Exmple3 [Left] and Exercise3[Right] assembly code

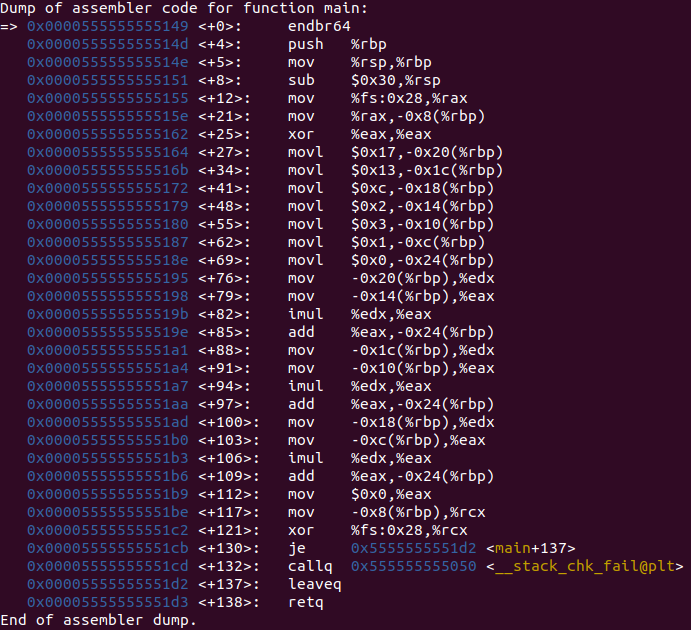


My theory is that this may have been due to my return statement at the end of the “Exercise3.c” program returning the y variable instead of 0. I made this change to the code and found a successful match for that line. The assembly code now matched near perfectly to the original assembly from Example3 with the exception of the sub value on line <+4>. With this I saved my code and continued on to writing the remainder of the report.

Exercise3.c final revision



Exercise3 final assembly output



1. **Testing**

Throughout this lab I conducted testing in a different manner than I had previously used. As outlined in the process section of this report I would slowly and methodically write code and check its assembly against the provided executable files assembly code. When I came across an unexpected value I would stop and ask myself why this would happen, what would cause this to happen, and what was happening in my code verses the executable. In this way my entire program was operating similarly to how I have utilized printf() statements in the past to output variables as they progressed through the program. I utilized the information I had available to me to make slight changes to my program in an attempt to match the executable assembly. When this wouldn’t work the code was reverted and a new tactic was attempted.

1. **Results**

The results of this lab successful in that each of the 3 Example executable files were reverse engineered with a high degree of accuracy. The most successful of the 3 was Exercise2 which produced and exact match from my code’s assembly output to the executable assembly code. If given more time and more resources I am certain I could make changes to the remaining 2 programs to increase their accuracy to the executable code, however I am confident in the accuracy of the code I have produced.

1. **Conclusions**

Based on the results and intent of this assignment I conclude that I have successfully accomplished the objectives of the lab. The focus of the lab, as stated in the intent portion of the report, was to practice reverse engineering executable software through a process that was provided to us. I utilized that process and produced 1 exact match and 2 near match assembly code results.

When I could not logically determine what was occurring in either the executable or the program I would prompt fellow classmates as to their interpretations of lines and their experiences with the assembly code. If it were not for my classmates I would have spent a significant amount more time troubleshooting my programs simply because of the line by line method in which I have been accustom to programming. It would have not occurred to me that the programs we were disassembling may be written more efficiently than how I had been used to writing code. This is another example of the value of having colleagues.

1. **References / Acknowledgements**

C Programming Language, B. W. Kernighan & D. M. Ritchie, 2nd Edition, Prentice Hall, 1988.

C Programming: A Modern Approach, K.N. King, Norton, 2008.

[X86 Assembly Guide](https://www.cs.virginia.edu/~evans/cs216/guides/x86.html)

Jared Larson, who suggested being more efficient with my code to produce different assembly