Project 4B: Reverse Engineering

Due Dec 13, 2021 by 11:59pm **Points** 50 **Submitting** a file upload **File Types** c
Available Dec 5, 2021 at 12am - Dec 24, 2021 at 11:59pm

Submission

✓ Submitted!
Dec 17, 2021 at 9:52pm (late)
Submission Details
Download moap_functions-1.c

Grade: 50 (50 pts possible)
Graded Anonymously: no

Comments:

See attached files.
zhiweiqian_314915_23574980_moap
ASMIT NAYAK, Dec 14, 2021 at 2:19am

See attached files.
zhiweiqian_314915_23574980_moap
ASMIT NAYAK, Dec 14, 2021 at 4:14pm

See attached files.
zhiweiqian_314915_23574980_moap
ASMIT NAYAK, Dec 15, 2021 at 2:37am

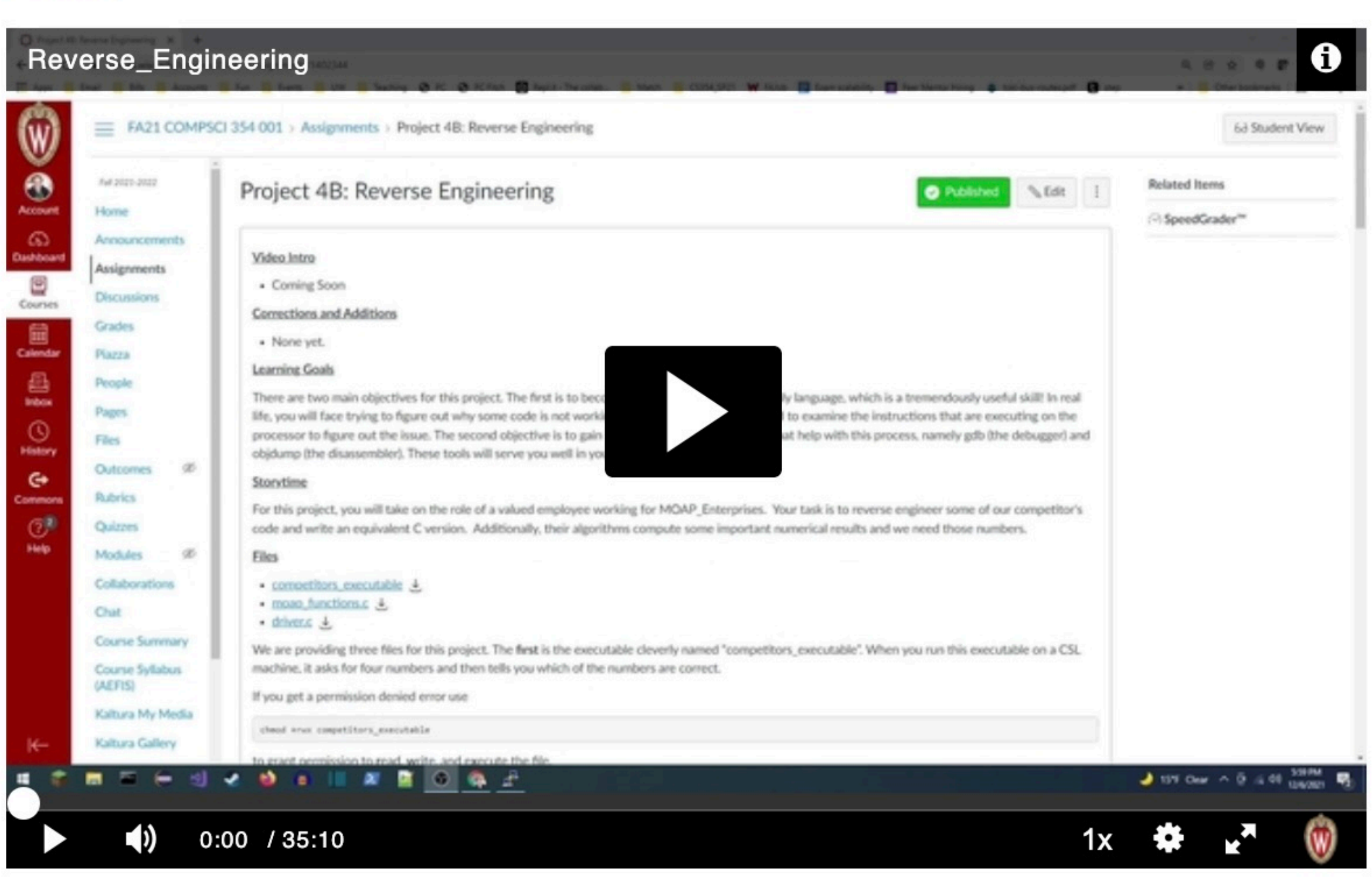
See attached files.
zhiweiqian_314915_23574980_moap
ASMIT NAYAK, Dec 15, 2021 at 2:29pm

See attached files.
zhiweiqian_314915_23574980_moap
ASMIT NAYAK, Dec 16, 2021 at 4:57am

See attached files.
zhiweiqian_314915_23574980_moap

This assignment was locked Dec 24, 2021 at 11:59pm.

Video Intro



Corrections and Additions

- None yet.

Learning Goals

There are two main objectives for this project. The first is to become familiar with x86-64 assembly language, which is a tremendously useful skill! In real life, you will face trying to figure out why some code is not working as planned, and you may need to examine the instructions that are executing on the processor to figure out the issue. The second objective is to gain familiarity with powerful tools that help with this process, namely gdb (the debugger) and objdump (the disassembler). These tools will serve you well in your future endeavors.

Storytime

For this project, you will take on the role of a valued employee working for MOAP_Enterprises. Your task is to reverse engineer some of our competitor's code and write an equivalent C version. Additionally, their algorithms compute some important numerical results and we need those numbers.

Files

- competitors_executable ↓
- moap_functions.c ↓
- driver.c ↓

We are providing three files for this project. The **first** is the executable cleverly named "competitors_executable". When you run this executable on a CSL machine, it asks for four numbers and then tells you which of the numbers are correct.

If you get a permission denied error use

```
chmod +rwx competitors_executable
```

to grant permission to **r**ead, **w**rite, and e**x**ecute the file.

The **second** file is moap_functions.c. This is the file you will do your work in, and this is the file you will turn in. This template file contains basic functions that take no parameters and return an integer. You will need to correct these functions after examining the disassembled executable.

The **third** file is a partial driver that we are providing to call your functions. You will need to fill in the lines for the function calls by reverse-engineering the function calls competitors_executable::main.

Compiling

Note we are #including "moap_functions.c" directly in driver.c

Compile your code using:

```
gcc driver.c -Wall
```

If you include both moap_functions.c and driver.c you will get a compiler error for multiple definitions of functions. It is generally considered bad programming practice to include .c files for this very reason. We are doing it here to hide the function prototypes so you have to figure out what they are as part of the assignment.

Turn in - There are two things to turn in for this project

1. moap_functions.c– upload this file to Canvas
2. The numerical results are computed by these functions and stored in variables r1 – r4. We are using the Canvas quiz feature to make the submission of these results easy for you to enter and easy for us to grade.

Reverse Engineering the Algorithms

Your goal is to reproduce the results of the algorithms - regardless of input you may write a different but equivalent algorithm. For example:

```
If the competitors_executable had a function:

int example_algorithm(int x) {
    if (x == 0) return 354;
    return 123;
}

And this function was called as:

int example_result = example_algorithm(0);

You could write:

int example_algorithm(int param1) {
    if (param1 !=0) return 123;
    else return 354;
}

These produce the same result.

Do not write!

int example_algorithm(int param1) {
    return 354;
}
```

There are many ways to write code that produces the same result for equivalent input. Another example: the competitors_executable.c may have used a while loop. You may choose to use a for loop, while loop, do while loop, or goto loop and get the same results. The exact assembly produced by your c code does not have to match the assembly generated by disassembling competitors_executable.

Getting Started / Strategy

1. Start by running competitors_executable. Try entering some numbers.
2. Disassemble the competitor's code
   - Use both gdb and objdump
   - Type this into the command line
     - objdump -d competitors_executable
     - objdump -d competitors_executable > disassembled.dump
       - to capture the output in a file and make it easier to read
     - gdb competitors_executable
       - start, run, continue, finish
       - list, list <how many lines>, or list <from_line> <to_line>
       - break <function_name> or *address_of_code
       - layout asm
       - stepi or nexti
       - info registers
       - x
       - help
3. Figure out the algorithm results r1-r4. Hint at several points these numbers appear in registers (which you can read using gdb) They are not necessarily the return values (found in %eax) because at least one function uses call by reference.
4. One function at a time
   - Find the first function call in main.
     - Use this block of code to identify parameters and return type.
     - Add the function call to driver.c
   - Find the code for the function call
     - Double-check your parameters and return type
     - Identify key features in the code and **label/comment** them
       - Prologue / Epilogue
       - Variable initializations (mov instructions to stack)
       - Jump instructions (Identify conditionals and loops)
       - Function calls
       - Return value preparation
     - Write the function in moap_functions.c
   - Compile with -S and compare your assembly to the disassembled code
   - Step through your executable in gdb and verify key intermediate results by comparing expected values in registers in your code and competitors_executable

Don't Panic!!! objdump

The first time you run objdump the output can be a little overwhelming. There is a lot of stuff here and most of it isn't important to this project. Some of it is setup before main is called, some of it is to restore the system after main returns, and some is related to the printf and scanf functions.

As you scan through the file for the first time, notice that it is divided into sections with headers "Disassembly of section..." Within those sections, the code is broken down by functions. The function names are in <>. Just search for the functions you need (Algorithm_1...4 and main)

Below are three columns: the first column contains the addresses where the instructions are stored in memory relative to a base address, the second block is the binary code for each instruction (ignore this), and the third column is the assembly instruction. The beginning of main is a little complicated to read in objdump due to the printf and scanf statements which are translated as memory addresses and not labeled as printf or scanf. Explore these with gdb next.

Don't Panic !!! gdb

The debugger, gdb, is an even more powerful ally in your search for clues. After you start the debugger, set some breakpoints. Breakpoints are places in the code where the debugger will stop running and let you take control of the debugging session. For example, a common thing to do will be to start up the debugger, and then enter:

```
    break main
```

to set a breakpoint at the main() routine of the program, and then type

```
    run
```

to run the program. When the debugger enters the main() routine, it will then stop running the program and pass control back to you, the user.

At this point, you will need to do some work. Type **layout asm** to switch to assembly layout where you can see the assembly instructions on the top window and enter your gdb commands in the bottom window. Use **stepi** to step through the code one instruction at a time and **nexti** to step through the code one instruction at a time but jump over function calls. Another useful command is info registers, which shows you the contents of the registers in the system. Use the examine command **x/x 0xADDRESS** to show you the contents at the address ADDRESS in hexadecimal. Note, the second x indicates the format (Hex), and the first x is the examine command. Use **help x** for more options. You can also have gdb disassemble the code by typing the disassemble command. Finally, **break *0xADDRESS** sets up another breakpoint at address ADDRESS. **Continue** resumes the execution until the next breakpoint is reached.

Getting good with gdb will make this project go smoothly, so it is worth spending a little time to learn how to use it. One thing to notice: using the keyboard's up and down arrows (or control-p and control-n for previous and next, respectively) allows you to go through your gdb history and easily re-execute old commands; if you are in the **layout asm** (i.e., assembly layout mode) only CTRL+p and CTRL+n will work to cycle through your history.

GDB Walkthrough

Looking at assembly code in gdb can be a little overwhelming the first time. The debugging tools were not enabled when the code was compiled. We'll have to examine the assembly version. Let's take a look first at Algorithm_1 and then at main. Start by setting a breakpoint with break Algorithm_1 then advance the code to Algorithm_1 by typing run. It will stop at the scanf line and ask you to enter 4 numbers. Just use 1 2 3 4 to continue – the program will tell you these are incorrect. The code will stop at Algorithm_1. At this point let's switch to layout asm to examine the assembly code. Gdb displays three columns in this mode. The first is the memory address of the instruction. The second is the relative offset from the first instruction in the function, and the third is the assembly code. Step through this function by typing **stepi**. You can push enter to repeat this command. At any time we can use info registers to get the contents of all of the registers as both hex and decimal values.

Next, let's take a look at main. Set a breakpoint with break main and restart execution of the program with run. Let's use **nexti** to advance through the beginning of main past the printf and scanf calls to the point where we call function f0. Remember that **nexti** will jump over functions, while **stepi** will step into functions. Push enter a few times to repeat **nexti**. Examine the instructions around <Algorithm_1>. These instructions prepare for the function call and clean up after the call. Pay special attention to what happens to the return value.

Hints

- All types are int, pointers to ints, or arrays of ints. The words char, short, and long do not appear in the original source code.
- All numerical results will appear in at least one of the registers several times.
- Neither your C code nor your assembly code must match exactly. It just needs to be in the spirit of code that does the same thing.
- Think carefully about function prototypes. Do they return a value? What parameters are passed to the function? Are they of type int or int*? What do these look like in assembly? Remember leaq is used to get the address of a variable.
- The compiler also uses leaq to do simple math. Remember the algorithm for multiplication by 3, 5, and 9 from the lecture?
- You will observe that parameters and local variables in the assembly code are referenced as offsets from %ebp. As you write your own version of the code, it may be useful to name your variables with a similar convention, something like int ebp4 = 3; or int *ebp16 = &ebp4.
- At least one of the functions is a void function and uses call by reference to return the result
- You may write any helper functions you wish to use
- Use both objdump and gdb. It's possible to only use one of these tools, but using both will save you many hours!

Style

Follow the style guide linked on the Canvas homepage.