

P3 Covert Ops

Due Apr 9 by 11:59pm **Points** 50 **Submitting** a file upload

Available Mar 30 at 12am - May 9 at 8am about 1 month

This assignment was locked May 9 at 8am.

Project 3: Covert Ops

Corrections and Additions

1. None yet.




Learning Goals

There are two main objectives for this project. The first is to become familiar with x86 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

In this project, you are a member of an elite Tech Ops team and will be assisting super spy Agent Storm as he attempts to rescue a group of captives from an enemy stronghold. To gain entry, Agent Storm will need to bypass several electronic locks. Fortunately, we have the compiled program that controls these locks. All we need to do is reverse engineer the executable and give the codes to our hero.

Files

- [Covert_Ops](https://canvas.wisc.edu/courses/230411/files/19265710/download?download_frd=1)  (https://canvas.wisc.edu/courses/230411/files/19265710/download?download_frd=1)
- [Codes.txt](https://canvas.wisc.edu/courses/230411/files/19265709/download?download_frd=1)  (https://canvas.wisc.edu/courses/230411/files/19265709/download?download_frd=1)
- [Covert_Ops_Template.c](https://canvas.wisc.edu/courses/230411/files/19265712/download?download_frd=1)  (https://canvas.wisc.edu/courses/230411/files/19265712/download?download_frd=1)

We are providing three files with this project. The first is the executable Covert_Ops. When you run this executable, it asks for four numbers and then tells you which of the numbers are correct. You can either enter the numbers manually or use the Codes.txt file with input redirection. Codes.txt is a text file containing four numbers. Unfortunately, the Codes.txt that we provided does not have the correct

numbers. Use it as a pattern for you to follow as you do your testing. To use the Codes.txt file, call Covert_Ops with input redirection:

```
./Covert_Ops < Codes.txt
```

Note that the original source code has been compiled with the following options. Use these same options when you compile your version.

```
gcc -o Covert_Ops Covert_Ops.c -m32 -fno-asynchronous-unwind-tables -fno-pie -fno-stack-protector
```

The final file is Covert_Ops_Template.c. This file contains the main function from the original C code.

Turn in

You will turn in two files. The first is Codes.txt with the correct door codes. The second is Covert_Ops.c. Your goal for this file is to write source code in C that performs the same task as the executable provided. There are many ways to write Covert_Ops.c that will reproduce the original meaning and generate the same results even though the exact lines of source code or assembly may be different. For example, `i++`, `i+=1`, `i=i+1`, and `i=1+i` all do the same thing; so does `addl $1, %eax`, and `incl %eax`. Don't worry about getting your assembled code to match exactly. It just needs to perform the same function. We will use this to verify that you understand the assembly code.

Getting Started

The first challenge is to figure out the codes expected by the Covert_Ops program. To do this, begin by running the provided code (with and without the Codes.txt file) to see what it does when you enter incorrect codes. Then use two tools: **gdb** and **objdump**. Both are incredibly useful for this type of reverse engineering work.

```
./Covert_Ops
```

```
./Covert_Ops < Codes.txt
```

`gdb` and `objdump` can be used to investigate the assembly code, memory, registers, and variables. These tools provide slightly different information, so be sure to examine both. Finally, write your own version of Covert_Ops, compile it with the same options we used and compare the assembly. Note, the assembly code does not need to match – it just needs to perform the same task!

I recommend just writing one function at a time. Use the following command to stop compilation after the generation of assembly code. You can use this output to examine the assembly code corresponding to your functions.

```
gcc -S Covert_Ops.c -m32 -fno-asynchronous-unwind-tables -fno-pie -fno-stack-protector
```

objdump

With objdump, two important command line options are:

- d, which disassembles a binary**
- s, which displays the full binary contents of the executable**

For example, to see the assembly code, you would type:

```
objdump -d Covert_Ops
```

This will show an assembly listing of each function. Your first task then might be to look at function `f0` and figure out what the code is doing.

The `-s` flag is also quite useful, as it shows the contents of each segment of the executable. This may be useful when looking for the initial value of a given variable or for the strings of text output by `printf`.

By redirecting stdout (i.e., standard output), you can capture the output of `objdump` in a file, such that you can look at this output without having to regenerate it every time. And, you can use both command line options (`-d` and `-s`) at the same time to create a complete dump of the contents of the executable as well as the disassembled contents.

```
objdump -d Covert_Ops > Covert_Ops.dump
```

Don't Panic!!!

The first time you run `objdump` the output can be a little overwhelming. There is a lot of stuff here that 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 `<>`. 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. Scroll down to function `<f0>` and start there. This one is the easiest. Search for `<f0>` to locate where the function is called in `main` and compare this to the template code. 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`. We'll explore these with `gdb` next.

gdb

The debugger, `gdb`, is an even more powerful ally in your search for clues. To run `gdb`:

```
gdb ./Covert_Ops
```

which will launch the debugger and prepare you for a debugging session. The command **run** causes the debugger to run the program, which will prompt you for input. However, before **running** the debugger, you likely need to 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 all 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. You can also have gdb disassemble the code by typing the **disassemble** command. Finally, **break *0xADDRESS** sets up another breakpoint at address ADDRESS, and **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.

Don't Panic Part 2!!!

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. So we'll have to examine the assembly version. Let's take a look first at f0 and then at main. Start by setting a breakpoint with **break f0** then advance the code to f0 by typing **run**. It will stop at the scanf line and ask you to enter 4 numbers. Enter 4 numbers separated by spaces and push enter. The code will again stop at f0. 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 <f0>. 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 codes will appear in one of the registers at least once.
- Neither your C code nor your assembly code has to match exactly. It just needs to be in the spirit of code that does the same thing. For example, there are lots of ways to write loops in C (for, while, do while, goto); you can't tell from the assembly what type of loop was in the C code. We will examine your code to see that you recognized that some kind of loop was used.
- Same thing for assembly code. Many of different versions of assembly can be written for any given C code segment. We do not expect the code to match exactly - just perform the same function.
- 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?
- 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 *ebp8 = &ebp4.

Style

1. Use meaningful variable names. Either use underscores for multi-word variables or CamelCase. Be consistent.
2. Be consistent with capitalization. For example, capitalize function names, use all caps for structs, and #defines, and use lower case for variables.
3. Indent to match scope. Use spaces instead of tabs.
4. Organize your code as follows:
 1. #include <>
 2. #include ""
 3. #defines
 4. Data Types (e.g., structures and typedef)
 5. Global variables – (use of global variables only when necessary)
 6. Function Prototypes
 7. Code - main() should be either first or last.
5. Comments
 1. Describe what each block of code is trying to do.
 2. Good variable names may make this description easier.

3. For functions describe the purpose of the function and indicate the purpose of input parameters, output parameters, and the return value.