# Project 4: Escape Room

## Corrections and Additions

1. When you enter the list of numbers, do not put commas between them. Only spaces.
2. 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 C. All we're looking for is that you recognize that some kind of loop was used.
   Same thing for assembly code. Lot's 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.
   So the things we're going to be checking when we grade include: function prototypes - return type and parameter types should match (but not necessarily variable names). If the function contains a loop of any kind then your code should also contain a loop - it doesn't need to be the same kind we used in the solution. Same with conditional statements. f1 contains a nested conditional - we're just looking to see if your function contains conditional statements that would result in the same return value. Same with function calls.

## Introduction

An escape room is a game in which a group of friends search a room for clues and solve puzzles with the goal of trying to escape the room. Combination locks are common puzzle elements presented to the players with clues to the combination hidden in the room. Today we're going to be searching some compiled C code for clues to the combination of a lock.

## Files

You will be provided with two files. The first is the executable Escape_Room. When you run this executable, it asks for six numbers and then tells you which of the numbers are correct. You can either enter the numbers manually or use the Combination.txt file. This is just a file with six numbers. Unfortunately, the Combination.txt file provided does not contain the correct combination, but you can use this as a starting point to change your guesses. To use the Combination.txt file call Escape_Room with input redirection:

```
    ./Escape_Room < Combination.txt
```

Note that the original source code (not provided) has been compiled with the following options:

```
    gcc -o Escape_Room Escape_Room.c -m32 -fno-asynchronous-unwind-tables -fno-pie -fno-stack-protect
or
```

- Escape_Room
- Combination.txt

## Turn in

You will turn in two files. The first is Combination.txt with the correct combination. The second is Escape_Room.c. Your goal is to reproduce the original source code from the executable binary. There are many to write Escape_Room.c that will reproduce the original meaning and result in the same binary executable after the compiler has operated on them. Don't worry about getting the exact version of the original source code. We're just going to use this to verify that you understood the assembly code.

## Learning Goals

There are two main objectives of this project. The first is to become familiar with x86 assembly language. This is a hugely useful skill! In real life, one is often faced with the task of trying to figure out why some code is not working as planned, and sometimes this leads to staring at the instructions that are executing on the processor to gain understanding. The second relates to the first: to gain some familiarity with powerful tools that help with this process, namely **gdb** (the debugger) and **objdump** (the disassembler). These tools will also serve you well in your future endeavors.

## Getting Started

The challenge is to figure out the combination expected by the escape room lock. To do this, use two tools: **gdb** and **objdump**. Both are incredibly useful for this type of reverse engineering work. First copy the executable to your own private directory and begin testing by running the code then use objdump and gdb to find the combination. Start by just running the code with and without the input file to see what the program does:

```
    ./Escape_Room
    ./Escape_Room < Combination.txt
```

Then use ojbdump and gdb to investigate the assembly code, memory, registers, and variables. Finally write your own version of Escape_Room, compile it with the same options we used and compare the assembly.

I recommend just writing one function at a time. C will let you compile a single function and does not require main to be present. To stop compilation after the generation of assembly code use the following command. You can use this output to examine the assembly code corresponding to your function.

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

Think carefully about the function prototype. Does it return anything? Are there parameters? How many? Does the function have local variables?

# 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 Escape_Room
```

This will show an assembly listing of each function. Your first task then might be to look at main() 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.

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 full dump of the contents of the executable as well as the disassembled contents. (Hint, hint!)

```
    objdump -d Escape_Room > Escape_Room.dump
```

# gdb

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

```
    gdb ./Escape_Room
```

which will launch the debugger and ready 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 first 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 fire up the debugger, and then

```
   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. You may type **layout asm** for switching to assembly layout where you can see the assembly instructions on the top window and you may type your gdb commands at the bottom window. One likely command you will use is **stepi**, which steps through the code one instruction at a time. Another useful command is **info registers**, which shows you the contents of all of the registers of the system. Another is **x/x 0xADDRESS**, which is the examine command. It shows you the contents at the address ADDRESS, and does so in hexadecimal. The second x determines the format, whereas 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 any breakpoint is reached again.

Getting good with gdb will make this project go smoothly, so spend the time and learn! 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 works. Getting good at using your history, whether in gdb or more generally in the shell you use, is a good idea.

See also the material from Friday July 3 about using gdb.

# 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.