gdb Lab
CSC 9010/5930 - Offensive Security
Grading: 10 points
Due Date: Jan 23rd, 2019 at midnight

Description: In this lab you will learn about the gnu debugger (gdb) and basic x86 assembly. Follow along with the instructions, write your answers to all questions in a separate `answers.txt` file, and submit on Blackboard when finished.

Part A:

1. Start up the Hacking textbook virtual machine and open a shell. Navigate to the directory `/home/reader/booksrc/` and find the file `game_of_chance.c`. Compile this file using gcc with the debugging flags on (-g command line option). Once you do this, you'll need to tweak the permissions for the executable by running the commands:
   `sudo chown root:root ./a.out`
   `sudo chmod u+s ./a.out`
   This will allow you to run and play the game of chance game, which you should do now to learn how it behaves.

2. **QUESTION:** What do the two commands in the previous step do?

3. Open the source file `game_of_chance.c` and trace the program execution starting at `main()`.

4. **QUESTION:** What memory segments will hold this code? What segments will hold the declared global variables? As you trace the execution of a specific game, what C construction is being used to store the current game?

5. Now start up gdb by typing `gdb a.out` to start the debugger and load the executable. You now have access to the interactive debugging interface. Type "run" to execute the program. This starts execution, but isn't very interesting until we add some breakpoints to stop the program flow. Quit the game now. You can add breakpoints at specific functions or line numbers by typing `breakpoint <number or name>` (you can also type "List" to see the source code if it was compiled with the "-g" flag). Try setting a breakpoint at the function `play_the_game` and running the program from the top. When you play the "Pick a Number" game now, the program should halt.

6. Step through the code using the "step" command, which executes a line of code or steps into a function call (as opposed to "next" which simply executes a function call and continues without stepping into the function). The debugger will show each line of code before it is executed. Step until line 285 is about to be executed. At any point, you can print the value of a variable. Try typing `print winning_number` to see what number you should pick (at this point, you can also use the `backtrace` command to see the current call stack).

7. You may now type `continue` to continue the game execution (and win the game with your inside knowledge). After you quit the game, type `quit` to leave gdb. Congratulations, you're a Gnu Debugger!

Part B:

1. Next, we need to be able to inspect the assembly code that goes with our program. Run an object dump of the game of chance executable with:
   `objdump -D a.out | head -20`
   This is assembly, but the notation looks unusual and there is a lot here to try to sift through. We will use gdb to help us more clearly examine the assembly. Load the program in gdb and type `set disassembly-flavor intel` to get more familiar assembly code notation. For any line or function, you can disassemble that particular section of code. Try `disass main` to see the main function in assembly.

2. **QUESTION:** What is the offset for the `jmp` instruction associated with the while loop in `main`? What is the offset for the `cmp` instruction where the command is compared to the constant value 7?

3. Again set a breakpoint at `play_the_game` and run the program until you hit that breakpoint. Now type `info registers` to see the contents of your processor registers.

4. **QUESTION:** What value is stored in the eip? The esp?

5. Now that we can see the assembly for different parts of the program, let's try inspecting the actual contents of memory. Inspect the top of the program stack by typing `x/x $esp` (note: The `x` command is short for "examine memory." If you want to examine more than one memory word, type the number you want after the forward slash, like `x/4x $esp`. You can also type a concrete memory address prefixed with "0x" instead of a register name, or change the format to binary or another size word by changing the second "x"). You can also inspect the base of the current stack frame using `x/2x $ebp`

6. **QUESTION:** What value is on top of the stack? What is the return address of the current stack frame? What does the return address point to?

7. We can also examine memory holding variables. Type `print player` to see the current player struct, then type `x/8x &player` to see the same data directly in memory.

8. **QUESTION:** What is the value of credits in decimal notation? How about hexadecimal? Where is this memory segment, lower or higher address than the stack?

9. If you wish to examine the flow of assembly code, you can use the `nexti` command to run the next assembly instruction in sequence (instead of the next source line of code).

Part C: Explore

1. Open the Kali 64-bit VM, download the executable from the course webpage, and run it to learn what it does and how it works. After you are comfortable with the program, load it into gdb and disassemble the main function *before you start the program.* From here, answer the following on your own.

2. **QUESTION:** What is the address of main?

3. **QUESTION:** Set a breakpoint and run the program. What is the address of main now? What happened?

4. **QUESTION:** Why do the addresses look different from the previous parts of the lab? Why are there more registers with different names?

5. **QUESTION:** Sketch out the stack frames for deposit() and main() during a deposit of the value 1

6. **QUESTION:**Which value in the previous example corresponds to the deposit amount of 1? What the value of that variable in hex? Why isn't it numerically `0x01`?

7. **QUESTION:** Continue running through various parts of the program, deciphering the control flow using the `List` command and any breakpoints you like. After you select "quit", some heap memory storing user structs will be freed. Step through this function and inspect the memory as you proceed through the loop. What happens to the data in memory after each struct is freed? Could this be a problem?

8. **QUESTION:** How helpful did you find today's lab exercise? What helped your learning the most? What would you change?

Rubric:
(10 points) submit your answer sheet on Blackboard.

Deliverables: Submit the answer sheet on Blackboard.