IC411 GDB Lab Name\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Fall AY2019

**Summary**: This lab will exercise the GNU Debugger (gdb), to introduce or refresh it for you. You will be stepping through two example programs.

**Expected Completion Time**: 1-1.5 hours

**Lab Weight**: 15 points

**Learning Objectives**:

- Introduce the x86 ISA, including its disassembly syntax and registers.

- Reinforce use of gdb debugger to examine code in execution.

**Collaboration**: Per course policy.

**Tasks**:

**string**

In this example, a program has been compiled with debugging symbols included. For example:

> gcc **-g** -o string string.c

There is a string embedded in thee program, but its value is only correct mid-program, and it never is sent to standard out. The string is obfuscated at the beginning and end of the program, but not in the middle. By debugging and examining the program, you'll be able to figure out the hidden string value.

- Save a copy of the string file to your working directory.

- Open or print for reference a copy of the gdb reference card.

- Before you do any debugging, look for plaintext clues in the executable!

> strings string

Did you find anything that obviously reveals the hidden string value? \_\_\_\_\_\_\_\_\_\_\_\_\_\_

- Set up gdb to use Intel syntax:

> echo "set disassembly-flavor intel" >> ~/.gdbinit

- Launch gdb, with the executable as an argument.

> gdb -q string

- Set a breakpoint in the main function:

(gdb) b main

- View the disassembly of the main() function:

(gdb) disass main

Although you won't understand all the x86 functionality, you may be able to identify some things that are going on:

- A loop is often identified with an increment of a variable value by one (0x1) using the add instruction. There are two loops in the program. Identify this kind of increment-by-one instruction in each loop, and copy them below:

- Local variables are stored on the stack below the base pointer, contained in the register 'rbp'. Since the stack grows down, local variables are *below* the base pointer in memory, such as [rbp-0x50]. give an example of an assembly instruction from main() that assigns a local variable its value using the mov instruction, which like a store instruction, copies a value to memory:

- Run the program, which will proceed until the breakpoint at main():

(gdb) r

- Get info about the local variables:

(gdb) info locals

What are the local variable names?

Is there any obvious meaning to the contents right now?

- List the contents of the C source code. Look for the line that starts the second loop.

(gdb) list 1,20

- Identify the source code line number that starts the second loop, and execute up to that point, with something like:

(gdb) until 14

- Now that the first loop is complete, let's look at local variable values again:

(gdb) info locals

Now that s1 is not obfuscated, what is its secret value?

- Looking at the source code, describe the means by which the string is obfuscated:

- Examine the registers:

(gdb) info reg

What is the value of the instruction pointer, relative to the start of main()?

Knowing that the current function occupies memory on the stack between the base pointer (rbp) and the stack pointer (rsp), both shown in hex, how many bytes are currently taken up on the stack by function main()? (hint: print $rbp - $rsp)

- Continue execution to the end:

(gdb) c

**pswd**

In this executable, there is a hard-coded password that you have to discover through debugging. This time, there will be no debugging symbols and source code included.

- Download the pswd executable to your local folder.

- As before, run 'strings' against the executable. Do you see any obvious hard-coded passwords?

- Launch gdb with pswd as the executable.

- Set a breakpoint at main().

- Try listing the source code again, and also try viewing a disassembly of main(). Which succeeds?

Why the difference this time?

- Run the program until the breakpoint at main(). Try viewing local variables. Does this succeed? Why or why not?

- Try viewing all the register contents. Does this succeed? Why or why not?

- Let's look for interesting function calls in the disassembly. Since we know this is a password checking program, let's look for a strncmp() or similar function:

(gdb) disass main

Look at the 'call' instructions, and find the one that references <strncmp@plt>. Note the address offset (in decimal) from main(). Since we are currently at a breakpoint at the start of main, we can set a break here as follows, omitting leading zeros:

(gdb) b \*0x400814 (substituting the address of the call in your code)

Now we can run the program to that breakpoint:

(gdb) c

You will be asked to enter your alpha code and the secret password.

- Calculate the size of memory occupied by the data stored inside the local area of main() on the stack. Use info reg to view the register contents, then subtract rsp from rbp. Subtract in hex, then convert to decimal. What is this value in decimal bytes?

- Now use the 'examine' function to examine a swath of memory beginning at the stack pointer (above which is our main function's local area on the stack, up to the base pointer). Use the offset value calculated previously, as the number of bytes to use. The 'c' indicates we'd like to see ASCII characters. Change 96 as needed to match your calculation.

(gdb) x/96c $sp

Notice anything? Scan until the password reveals itself. Confirm by quitting gdb, re-running the executable by itself, and trying the password. It will end in '\n', in this example. What's the password?

Write the alpha code you entered:

Write your 'Magic Key':

**Submission**: this paper in hardcopy with completed answers.