GDB Overview Lab

Adopted and Modified from Professor Soraya Abad-Mota

CS 341 - Spring 2021

1 Examining Binary Programs

First, you need to obtain the file to use, it is called someFile.txt, it is on the course website with this assignment.

The extension (.txt) claims it is a text file, but is it? To learn what type of file it is, the unix command file can be used. When you run file for this file, it should give you this information: x86 ELF-format exectuable, not a text file, despite of its extension. How many bits was this program compiled for? ELF (Executable and Linking Format) is the basic format of binary files on most modern systems (Macs are an exception). Obviously, opening (running!) this program without knowing what it will do would be a mistake. Linux contains a wide array of programs for manipulating and examining binary files, with objdump being the most powerful.

A program called: strings can be run on the file to get a list of the text strings in the binary. Anything here that gives you pause? Similarly, try dumping the contents of the .rodata (read-only data section) using objdump -j .rodata -s someFile.txt. Note how this format gives you both the binary contents and interpretation as a string of this section. man objdump will give you much more information about the capabilities of objdump.

Finally, disassemble the program using objdump -d someFile.txt. You'll see a large number of procedures that get disassembled. Almost all of these are support functions supplied by the system libraries to get the program running. The instructions under the symbol are the ones you'll probably care about. Knowing exactly what it's doing from looking at the assembly is a little difficult. As an aside, you may obtain the ascii codes table by doing man ascii in any unix-based system. When you finish this section, you should have copied the output produced by file, strings, and objdump in a text file to be submitted with this assignment.

Break down of section 1 commands you will need to show for your submission:

1. **file** - The extension (.txt) claims it is a text file, but is it? To learn what type of file it is, the unix command file can be used.
2. **strings** - can be run on the file to get a list of the text strings in the binary.
3. **objdump -j .rodata -s someFile.txt** - show what objdump does here
4. **objdump -d someFile.txt** - Try it with different flags
5. **gdb** - open in gdb

2 Running Binary Programs in the Debugger

Next, open the file in gdb; GDB provides many of the same tools that the command line provides, while also allowing you to execute the program step by step. Unfortunately, because we don't have source code, we cannot actually set breakpoints or list source code like we normally do in the debugger. We have to work at the machine code level.

Start by trying the following commands in GDB:

1. **print main** - Note that GDB saves the result of this command, the address of main, in the variable $1. This lets you use it later.
2. **break 1** - Sets a breakpoint at the start of main.
3. **run** - Start the program. Note that we're assuming the program starts with main; a somewhat more clever program would actually have the program start elsewhere!
4. **info reg** - Print the program's registers. You can print register contents using the GDB print command by prefixing the register name with a dollar sign. Note that the current program counter is called the "instruction pointer" in x86-speak, and is shown in GDB as register $eip for a 32-bit program and in $rip for a 64-bit program.In the

following commands, use the appropriate register, depending on the number of bits the program was compiled for.

5. **disassemble $eip** - Disassemble the program around the current instruction pointer.

6. **display/i $eip** - Always print the disassembly of the current instruction pointer, which is the next instruction that will be run.

7. **stepi** - Step a single assembly instruction.

3 Reverse Engineering The Program

Hopefully you have read sequentially until this point and have followed the steps suggested above. The program we have given you contains the first phase of a bomb. The last task in this lab is to figure out what this first phase of the bomb is doing and find the key that difuses the bomb. We are not keeping a score at this time, so if you explode the bomb in the process, it will not affect your grade, but you might want to practice setting up the appropriate breakpoints so that does not happen.

The process we suggest is to start by entering the GDB commands described above. Then, step instruction by instruction through a candidate function, looking at the upcoming instructions, and examine memory and registers as the program runs. This should give you a general sense of what a program is doing.

Setting breakpoints in strategic places (you can set breakpoints at absolute memory addresses in the instruction stream as well!) can also help you when working with binary programs. In particular, if you try to single step through the whole program (with stepi), that would take forever. Instead, whenever the program is about to leave main by calling another function (e.g. the next displayed instruction is a call instruction), set a breakpoint at the next assembly instruction after the call instruction, and run continue (or simply c) to have the program run until it hits the next breakpoint.

Another gdb feature you may not have seen are watchpoints. Watchpoints are breakpoints that trigger whenever a memory location changes; note, however, that they work best on global variables and can be tricky to use on local variables. They're useful when you want to find the code that's modifying a particular variable, especially the code that shouldn't be modifying that data (e.g. from memory errors).

If you can find the string that defuses the bomb, save it in a text file and add to this the explanation of the piece of assembly code where the bomb is defused. The string alone without the explanation will not give you all the credit in this portion of the lab. If you cannot find the defusing string, at least pick a piece 2 of assembly code in this program and explain what it is doing to the best of your understanding. If you want to get more experience with GDB, feel free to write your own programs, compile them, and use the same tools to examine programs where you know what they do. (Note that you don't need to do this for the lab exercise itself, just do it to get more insight in assembly code.)

4 Submission Details

In order to get credit for this lab, submit the two files described below by the due date indicated on Learn.

1. Save in a text file the output of your explorations, indicated in sections 1 and 2; give the name **labGDB.txt** to that text file.
2. Save the string that defuses the bomb and the explanation in another text file, call it **defuseGDB.txt**. If you cannot find the defusing string, **pick at lease 2 pieces** of assembly code in this program and explain what it is doing to the best of your understanding.
3. Submit both files to Learn in the place for this lab.