8 Assembly-Level Debugging with gdb

The GNU Debugger (`gdb`) is an incredibly useful tool for debugging C programs. It lets you step through a program one program statement at a time, inspect local variables, set breakpoints, and so forth. But `gdb` is also a big time-saver when working with assembly code. Spending a little time getting to know `gdb` will take all the guesswork out of developing an exploit for a vulnerable program.

8.1 Disassembly mode

When you start gdb with a program, e.g.,

```
$ gdbtui myprogram
```

you can switch it into "disassembly mode."

```
(gdb) layout asm
```

Your source code window in gdb will now be filled with assembly code.

8.1.1 Debug symbols

This note is primarily for people who already have a little experience using gdb.

gcc lets you add what are called "debug symbols" for debugging a program. These symbols are handy when debugging C code, but not all that useful when debugging at the assembly level. If you've used gdb before, you may be in the habit of using this flag.

Try generating the assembly for a program with `gcc -S` and then generating assembly for the same program with debug symbols using `gcc -g -S`.

For a simple "hello world" program, I get 34 lines of assembly using the first option and 220 (!!!) lines of assembly for the second version. What's going on? In short, gcc generates lots of supporting information to help gdb do its job. This is very useful when debugging C code, and all of this extra information is hidden from you at the source code level. But when debugging at the assembly level, it adds a lot of unnecessary noise.

I suggest that you *do not* use the `-g` flag when generating assembly code for this class.

8.2 *Running programs*

Running a program in `gdb` is easy. At the `(gdb)` prompt, type:

```
(gdb) run
```

Note that, although you can run programs this way, this method may not be all that useful for Lab 5. Instead, you probably want to run a program with some input from `STDIN`. You will also want to know how to pause a program at a given point, which is called *setting a breakpoint*.

8.3 *Running programs that read from STDIN*

Using `gdb` with programs that read from `STDIN` is a little tricky because the `gdb` does not attach the program's `STDIN` to your terminal's `STDOUT`. `gdb` is using *your* `STDOUT` to control `gdb` itself. Fortunately, if you save your desired input into a file, you can ask `gdb` to pass that input along to the program:

```
(gdb) run < myfile
```

8.4 *Setting assembly breakpoints*

It is often very useful to pause the execution of a running program so that you can inspect its state (local variables, call stack, etc.). A *breakpoint* is a location at which you ask `gdb` to pause. In `gdb`, you can set breakpoints at both the source code (e.g., C) level or at the assembly level. We will primarily want to set assembly breakpoints in this class.

Setting an assembly breakpoint is done by using the address of the instruction that you want `gdb` to "break" at. E.g.,

```
(gdb) break *0x80483d4
```

You can also type the shorthand:

```
(gdb) b *0x80483d4
```

If you're using `gdbtui`, you should see a `b+` appear in the assembly listing at the location you requested.

The `*` in the command above is mandatory; it tells `gdb` to interpret the argument to `break` as an address and not as a label (which is the default).

Note that if you supply a label (e.g., a function name), the breakpoint will appear *after* the function's prologue. This may not be what you want! E.g., suppose I have the function:

```
0x80483d4 <main>               push    {fp, lr}
0x80483d8 <main+4>             add     fp, sp, #4
0x80483dc <main+8>             mov     r1, #2
```

and I call `break main`. Then the breakpoint will be set at `main+8`, at address `0x80483dc`.

8.5 *Inspecting registers*

You can look at the state of all of your registers using:

```
(gdb) info registers
```

You can also inspect a single register by giving the above command the name of a register:

```
(gdb) info registers r0
```

8.6 *Stepping, stepping over, and continuing*

As when debugging C code in `gdb`, you can step to the next instruction when in assembly mode. Note that the ordinary `step` command steps *to the next C statement*. A single C statement can correspond with many assembly instructions. Instead, to step at the assembly level, use

```
(gdb) stepi
```

which steps to the next assembly instruction. Alternatively, you can also use the shortcut,

```
(gdb) si
```

It's also worth noting that pressing Enter will repeat the last command you ran.

You can also "step over" branch instructions (like `bl`) just like you might step over functions (using `next`) with:

```
(gdb) nexti
```

or the shorthand

```
(gdb) nexti
```

Finally, if you've set an assembly breakpoint, and you want to continue until your breakpoint is hit, use:

```
(gdb) continue
```

or the shorthand

```
(gdb) c
```

8.7 *Printing values*

You can print the values of registers and memory locations. This is very useful in combination with `gdb`'s formatting options.

For example, you can print the register `sp` like so:

```
(gdb) p $sp
```

Would you like to see the output in hexadecimal?

```
(gdb) p/x $sp
```

Why might this be preferable to `info registers ssp`? For starters, you can use it to print mathematical expressions, like:

```
(gdb) p/x $sp - 32
```

In fact, if you know that a value is a pointer, you can tell `gdb` to "cast" the value, which is very useful for understanding what data exists at certain memory locations. For example, the following expression dereferences (the first `*`) the `int *` (cast) stored at location `$sp - 32`.

```
(gdb) p *(int *)($esp - 32)
```

Or maybe you want to see that in hexadecimal?

```
(gdb) (gdb) p/x *(int *)($esp - 32)
```

8.8 *Inspecting values*

Perhaps you would like to inspect a word stored at a given location, but you'd like to view it one byte at a time, in hex format? Suppose our word starts at `0xabcdef01`:

```
(gdb) x/4xb 0xabcdef01
```

The `x` command asks to *examine* memory. The arguments to `x` are after the / and are *number*, *format*, and *unit*. The above command examines "4 bytes, each printed in hexadecimal."[1]

[1] A complete reference for examining memory can be found at `https://web.mit.edu/gnu/doc/html/gdb_10.html#SEC58`.