The GNU debugger gdb proves highly useful when analyzing memory safety vulnerabilities, and is well worth the time spent to become comfortable with it. In this tutorial we introduce basic features that facilitate the development of exploits.

A basic gdb workflow begins with loading an executable:

```
gdb executable
```

Alternatively, if the executable takes argument you may as well use:

```
gdb --args executable arg1 arg2 ...
```

You can then start running the program with:\(^{1}\)

```
$ run [arguments-to-the-executable]
```

In order to stop the execution at a specific line, set a breakpoint before issuing the “run” command. When execution halts at that line, you can then execute step-wise (commands next, step, stepi) or continue (command continue) until the next breakpoint or the program terminates.

```
$ break line-number or function-name
$ run [arguments-to-the-executable]
$ step        # branch into function calls
$ stepi       # execute next instruction only
$ next        # step over function calls
$ continue    # execute until next breakpoint or program termination
```

\(^{1}\) In this tutorial we have changed gdb’s default prompt of (gdb) to $.
Once execution stops, you will find it useful to look at the stack backtrace and the layout of the current stack frame:

$ backtrace
$ info frame 0
$ info registers

You can navigate between stack frames using the up and down commands. To inspect memory at a particular location, you can use the $x/FMT$ command:

$ x/16 $esp
$ x/32i 0xdeadbeef
$ x/64s &buf

The FMT suffix after the slash indicates the output format. Other helpful commands are disassemble func to look at the assembly of a function, and info symbol addr to show the symbol name at a given and the section where it is located. You can get a short description of each command via

$ help command

During exploit development, you will find it useful to modify the value at an arbitrary address, e.g., to manually alter control-flow and execute injected code. For example, assuming a breakpoint fires in the current function and you would like to overwrite the address of the return instruction pointer (RIP):

$ i f 0  # shorthand for "info frame 0"
Stack frame at 0xbffff6e0:
eip = 0x8048e2f in foo (foo.c:42); saved eip 0x8048e2f
called by frame at 0xbffff6f0
source language c.
Arglist at 0xbffff6d8, args:
Locals at 0xbffff6d8, Previous frame’s sp is 0xbffff6e0
Saved registers:
ebp at 0xbffff6d8, eip at 0xbffff6dc

Here, you see that the RIP has the value 0x8048e2f and sits on the stack at address 0xbffff6dc. To change the value of the RIP, you can do this:

$ set *0xbffff6dc = 0xbfffffff  # give RIP new value
$ x/i *0xbffff6dc  # show instruction at new RIP
You may find it helpful to dump the memory image ("core") of a program that crashes. The core captures the process state at the time of the crash, providing a snapshot of the virtual address space, stack frames, etc., at that time. You can activate core dumping with the shell command:

```
% ulimit -c unlimited
```

A crashing program then leaves a file core in the current directory, which you can provide to the debugger together with the executable:

```
gdb executable core
$ bt # same as "backtrace"
$ up # move up the call stack
$ i f 1 # same as "info frame 1"
$ ...
```

When debugged a program that spawns new process, `gdb` does not step into the spawned process by default. In case you write an exploit that wraps the original program by calling `execve`, it comes in handy if `gdb` “follows” the new program. To do so, you need to adjust the `follow-fork-mode` setting, which defaults to the value `parent`:

```
gdb -e foo -s bar # load executable foo and symbol table of bar
$ set follow-fork-mode child # enable debugging across programs
$ b bar:f # breakpoint at function f in program bar
$ r # run foo and break at f in bar
```

There exists also an alternate display mode in `gdb`, TUI mode, which uses `ncurses` to render multiple panes with contextual information (e.g., assembly, source, registers) in the same terminal window. You can enter TUI mode either via the keystrokes `ctrl+X A`, or by issuing the `layout` command from within `gdb`. To display assembly and source at the same time, use the command `layout split`.

Finally, you can find a concise summary of all `gdb` commands at:

```
```

Now get your hands dirty...