GDB Intro

January 28, 2016

1 GDB Intro

Matthias Vallentin has written an introduction to GDB that might be useful when you begin your exploits. **Note that you will be using the `invoke -d`, not the `gdb` command, for this project, as explained by the previous section.** 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]
```

Sometimes, as in the project with `egg` and `exploit`, a program requires input to be piped, or streamed. In that case it doesn't take arguments, but you can still run it with input. For example:

[1] In this tutorial we have changed `gdb`'s default prompt of `(gdb)` to `$`.

```
(from the shell)
$ ./egg > input.txt
$ gdb dejavu
(gdb startup message)
$ break 8
$ run < input.txt
```

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
```

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 = 0x8048412 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 also find it useful to check the contents of the environment variables:

```
$ x/s *((char **)environ)
```

Finally, you can find a concise summary of all `gdb` commands in the gdb-refcard.pdf at:

https://drive.google.com/file/d/0B0Xho2LbahtreUktV1huYk1fSHc/view

Now get your hands dirty...