Reverse Engineering with GDB

2LT Neil Marklund
17X
CyBOLC 18-503
8 February 2019

IDA vs GDB

- Command line tool
- Debugging tool as well as disassembler
  - means that you can do static and dynamic analysis
- Free and open-source

- GUI
- Disassembler only (but has support for different processors)
- Has free version with limited features. Full version starts at $589.

Debugging

- gdb allows a user to set breakpoints, or points in memory where the program will pause execution
- This allows the user to examine memory locations within the program before going on to the next step

Debugging Commands

Start gdb:

```
user:~$ gdb --args <program> [command_line_args]
user:~$ gdb --args ./add 1 5
user:~$ gdb ./hello # this is fine if program does not require arguments
```

Run (or restart) program:

```
(gdb) run # r for short
```

Set breakpoint:

```
(gdb) break *<memory_addresss>
(gdb) b *0x400add
(gdb) break <function_name>
(gdb) b main
```

Delete breakpoint:

```
(gdb) clear *<memory_addresss>
(gdb) cl *0x400add
(gdb) clear <function_name>
(gdb) cl main
```

Quit gdb:

```
(gdb) quit
```

Stepping Through Program

Step (equivalent to one line of source code)

```
(gdb) step [number_of_lines] # 1 by default
(gdb) s
```

```
int add(int a, int b) {
    return a + b;
}

int main() {
=>    add(5, 6);
      return 0;
}
```

```
int add(int a, int b) {
=>    return a + b;
}

int main() {
       add(5, 6);
       return 0;
}
```

Step Instruction (step to next assembly instruction)

```
(gdb) stepi [number_of_lines] # 1 by default
(gdb) si
```

```
=>  0x0000000000400aae <+0>: push   rbp
    0x0000000000400aaf <+1>:    mov    rbp,rsp
```

```
    0x0000000000400aae <+0>:  push   rbp
=>  0x0000000000400aaf <+1>:  mov    rbp,rsp
```

Stepping Through Program

Next (same as step, but will skip over function calls)

```
(gdb) next [number_of_lines] # 1 by default
(gdb) n
```

```
int add(int a, int b) {
    return a + b;
}

int main() {
=>    add(5, 6);
      return 0;
}
```

```
int add(int a, int b) {
      return a + b;
}

int main() {
      add(5, 6);
=>    return 0;
}
```

Next Instruction (same as stepi, but will skip over function calls)

```
(gdb) nexti [number_of_lines] # 1 by default
(gdb) ni
```

```
=> 0x0000000000400ad0 <+14>:  call   0x400aae <add>
   0x0000000000400ad5 <+19>:  mov    eax,0x0
```

```
   0x0000000000400ad0 <+14>:  call   0x400aae <add>
=> 0x0000000000400ad5 <+19>:  mov    eax,0x0
```

Stepping Through Program

Continue execution (used when program reaches breakpoint)

```
(gdb) continue
(gdb) c
```

Memory Examination

- After pausing program execution, gdb allows the user to examine contents of memory locations
- Ability to display the same memory location in different forms (hex, decimal, strings, etc.)
- Can also look at the contents of registers (rip, rbp, rax, etc.)

Memory Examination Commands

examine memory

```
# default form will be a word (4 bytes) in hex
(gdb) x <memory_address> # notice no asterisk is needed before addresses here
(gdb) x 0x492124
(gdb) x $<register> # bash-like variables for common registers
(gdb) x $rip
```

display memory in different form

```
(gdb) x/[length][size_modifier][format] # length is 1 by default
```

Format Options
x - hexadecimal
d - decimal
u - unsigned decimal
c - char
i - instruction
s - string

Size Modifiers
b - byte
h - half word (2 bytes)
w - word (4 bytes)
g - giant word (8 bytes)

Memory Examination Commands

Examples

```
char array[] = "01234ABCDEFG";

(gdb) x/d array # Display in decimal
0x7fffffedeeb: 48
(gdb) x/x array # Display in hex
0x7fffffedeeb: 0x30
(gdb) x/s array # Display in string
0x7fffffedeeb: "01234ABCDEFG"
(gdb) x/i $rip  # Translate opcode to assembly instruction
=> 0x400b82 <main+53>:  mov    rdx,QWORD PTR [rbp-0x8]
(gdb) x/2x array # Display 2 bytes in hex
0x7fffffedeeb: 0x30    0x31
(gdb) x/2hx array # Display 2 halfwords in hex
0x7fffffedeeb: 0x3130  0x3332
(gdb) x/wx array # Display 1 word in hex (notice endianness flips order)
0x7fffffedeeb: 0x33323130
```

Memory Examination Commands

display content in all registers

```
(gdb) info registers [specific register] # no dollar sign required here
(gdb) i r eax
```

display local variables of current function

```
(gdb) info locals
```

look at important values regarding the current stack frame (current function’s address space)

```
(gdb) info frame
```

Disassembly

- gdb also allows static analysis of programs through a disassembler
- Process of translating machine code to assembly code

Good Resource: https://godbolt.org/

Disassembly Commands

Disassemble

```
(gdb) disassemble [function_name] # default main
(gdb) disas add
# From Command Line
user:~$ gdb -batch -ex 'disassemble [function_name]' <file_name>
user:~$ gdb -batch -ex 'disassemble add' main.out
```

Change disassembly syntax

```
(gdb) set disassembly-flavor <att|intel> # default att
(gdb) set disassembly intel
```

In order to save settings, you can write them in a `~/.gdbinit` file

```
# example ~/.gdbinit file

set disassembly-flavor intel
set pagination off # so you don’t have to press enter to scroll
```