98-212

GDB

Why learn GDB?

- Static analysis is great, but sometimes we need to poke and prod.
- Check assumptions
  - I *think* this will be at this address at this point in time.
- Check that your exploit is doing what you think.
- Develop shellcode
  - It won't work the first time.
  - Or the second time.

Basics

- man gdb -- seriously :D
- Pass "-g" to gcc for debug symbols.
- Loading and running program
  - gdb $BINARY_PATH
    - (gdb) r
    - (gdb) r --prog-flags
  - gdb --args $PROGRAM_NAME --prog-flags
    - (gdb) r (flags set from gdb invocation)
  - gdb (NO ARGS)
    - (gdb) file EXECUTABLE
    - (gdb) symbol SYMB_FILE

'p'rint

- Print variables in scope
- Can also be used as a calculator (think pointer arithmetic)
- Most useful for actually debugging when you have symbols.
- p EXP
  - p VARNAME
  - p (0x500 + 4*81)
  - p $eax
  - p *(ADDR)

'b'reakpoints

- A debugger is pretty useless without breakpoints.
- Tell debugger where and under what conditions to stop execution.
- b FUNCTION_NAME (need symbols)
- b FILE_NAME:LINE_NUM
- b *ADDR
  - b 0xc0000000
  - b *$edi
  - b +/- NUM -- offset from current position

'b'reakpoints

- We can set conditions
  - b EXPR if i == 9001
  - cond BP_NUM i == 9001
- clear FUN_NAME|*ADDR|F_NAME:LINE
  - Remove all breakpoints at that location
- List breakpoints with info b
  - delete BP_NUM
- tbreak -- deleted when hit first time
- disable/enable BP_NUM

e'x'amine

- Allows us poke at memory
- Format is x/FMT ADDR
  - FMT -- COUNT:FORMAT:SIZE
    - 4xw (4 words as hex)
    - 2s (2 strings)
    - 3i (3 instructions
  - ADDR -- Expression
    - 0xc0000000
    - $eax
    - ($esi + 4 * $esi)
    - *(ADDR)
- disassemble ADDR -- print assembly

'i'nfo

- Poke at environment in which we're debugging.
- info registers -- dump registers
  - shorthand: i r
- info args -- dump args to curr func
  - shorthand: i ar
- info stack -- dump call stack
  - Can you guess the shorthand for this?
  - where and bt do the same thing
- info threads, info scope, info locals, etc
- info file (find entry point)

Call Stack

- up N, go up N stack frames and start poking around
- down N ...
- frame N -- get number from bt or info stack
- info frame -- dump rip, saved rip, saved registers, etc.

Controlling Execution

- You hit your breakpoint, now need fine-grained control.
- 'c'ontinue -- continue to next bp
  - Pass count to skip over next N bp's
- 's'tep -- step over next line of code
  - Takes count also, needs source.
- stepi (si) -- step over asm instead
  - No source needed!
- next & nexti -- same as step variants, but they don't go into functions

Controlling Execution

- until
  - Takes same arguments as break, but must be within current stack frame
- finish
  - run until current stack frame ends
- return VAL
  - return to caller with VAL
- jump LINE or *ADDR
  - Jump to location, skipping instructions between current location and target location.
  - Use within current stack frame only.

Display

- Usually we have specific set of state we look at when breaking. We can automate this.
- display/4i $rip, etc.
  - Runs everytime execution stops.
- enable/display
- info display
- undisplay DISP_NUM

Misc

- ADDR@LEN -- Display as array
  - p/d 0x800c000@20
- {type}ADDR -- Display as type
  - Useful for structs
  -
- whatis EXP -- gives type
- ptype TYPE
  - Print out type definition
- info address SYMBOL
  - Get address of symbol (need symbols)

Misc

- set x = 20
  - If x is symbol in program, this changes state of program.
- set env LD_PRELOAD ./malloc.so
- set logging [on/off]
- TUI
  - tui reg general
  - C-x C-a to close.
  - Handy, but I've noticed issues
- gdb --core CORE_FILE
- gdb attach PID

Special

- GDB can debug remote targets
  - connects to gdbserver
  - qemu-{arm,mips,...} supports this.
  - gdb remote target localhost:1234
- reverse execution
  - First target record
  - Go forward (step, next, etc)
  - Then backwards
    - reverse-step
    - reverse-next
    - set exec-direction [forward/reverse]