GDB Tutorial : *A trip down memory lane*

GDB Basics - Invoking GDB

gdb program

- The basic form, loads program and waits for your commands

- **gdb *program core***
  - Loads program and the associated core file

- ***gdb program* -p *pid***
  - Loads program and attempts to attach to the currently executing program with a PID of *pid*
  - One can find the pid of a running process via
    - pgrep -l *program_name*

GDB Basics - Exiting GDB

- **quit**
  - If attached to a target, unattaches from the process then exits
  - If running a target, kills the target then exists
- **kill**
  - Kills the running target
- **signal *number***
  - Sends the specified signal to the target being debugged

GDB Basics - Invoking the Target

Pro Tip: You can substitute r for run!

- **run**
  - Begins the execution of the target
- **run *argument1 argument2 ...***
  - Begins the execution of the target with the arguments
- **start**
  - Inserts a temporary breakpoint at the beginning of the main procedure then runs the target
- **start *argument1 argument2 …***
  - Inserts a temporary breakpoint at the beginning of the main procedure then runs the target with the arguments

GDB Basics - Managing the Target

- **info program**
  - Displays miscellaneous information about the execution state of the target
- **info variables**
  - Displays a list of all global variables and their locations
  - Note that this can take quite a while on a large program
- **info shared**
  - Displays a list of all shared object files that have been loaded into process memory and the address spaces onto which they are mapped
- **info threads**
  - Displays a table of currently executing threads
- **info registers**
  - Displays the contents of certain registers
- **info source**
  - Displays information about the source files used in the current target

GDB Basics - Setting Breakpoints 1

A breakpoint makes the program stop whenever a certain point in the program is reached

Pro Tip: You can substitute b for break!

- **break**
  - Inserts a breakpoint at the current point of execution
- **break location**
  - Inserts a breakpoint at the specified location
  - Note that location can be specified in several ways, most importantly using function names or filename:line
    - break main
      - Sets a breakpoint at the beginning of the main function
    - break main.c:100
      - Sets a breakpoint at line 100 in the main.c file (or as close as it can get)

GDB Basics - Setting Breakpoints 2

- **break location if *condition***
  - Inserts a breakpoint at the specified location but the breakpoint only triggers when the specified condition is met
- **tbreak**
  - Can be used in all the same ways as break but the breakpoint inserted is temporary and will be removed after it is hit
  - Useful for creating temporary breakpoints
- **rbreak *regex***
  - Sets a breakpoint at all locations matching the given regex expression

GDB Basics - Managing Breakpoints

- **info break**
  - Displays a list of breakpoints indicating among other things their locations, conditions, and enabled state
- **disable/enable**
  - Disables/enables the current breakpoint
- **disable/enable *n***
  - Disables/enables the breakpoint numbered n
- **clear**
  - Deletes any breakpoint at the next instruction to be executed in the selected stack frame
- **clear *location***
  - Deletes any breakpoints at a given location
- **delete *n***
  - Deletes the breakpoint numbered n
- **continue**
  - Resumes execution of the target
- **skip *n***
  - Skips the next n (default to 1) crossings of the breakpoint

GDB Basics - Watchpoints

Watchpoints cause execution to stop when the value of a given expression changes.

- **watch *expression***
  - Sets a watchpoint for when the value of an expression is written into by the target
- **rwatch *expression***
  - Sets a watchpoint for when the target reads the value of an expression
- **awatch**
  - Sets a watchpoint when the target reads or writes the value of an expression
- **info watch**
  - Displays a table of the current watchpoints including location and status
- **Watchpoints can be enabled/disabled/cleared like breakpoints**
- **Pro Tip**
  - You can create a watch point at a given address (assuming you know the type) as follows
    - watch/awatch/rwatch *(int*)0xbaadf00d

GDB Basics - Catchpoints

Catchpoints cause the debugger to stop the target when certain program events occur

- **catch *event***
  - The type of event can vary depending on the code type (C++ vs ADA)
    - throw ***regex*** - When a c++ exception is thrown
    - rethrow ***regex*** - When a c++ exception is rethrown
    - Catch ***regex*** - When a c++ exception is caught
    - Exception ***name*** - When an ADA exception occurs
    - exception unhandled - an unhandled ADA assertion
    - assert - a failed ADA assertion
    - Exec - a call to exec
    - syscall ***name*** - a particular syscall
    - fork/vfork - fork or vfork are called
    - load/unload ***regex***
    - signal ***signal***
- **info break**
  - Displays a table of the current catchpoints
- **Catchpoints can be disabled/enabled/cleared/deleted like breakpoints**

GDB Basics - Stack Frames

- **bt**
  - Displays the current function stack for the current thread
  - Optionally you can specify bt full and it will show the arguments and local functions in the current frame
- **where**
  - Displays information about the currently selected stack frame
- **info locals**
  - Displays the variables that are local to the current stack frame
- **frame *number***
  - Changes the current stack frame to the frame number indicated
- **up/down**
  - Moves up/down a single frame in the current stack

GDB Basics - Stepping 1

- **finish**
  - Resume execution of the target until it hits the end of the current stack frame
- **step or s**
  - Steps to the next source line
  - Think of this like the visual studio function step in
- **next or n**
  - Steps to the next line of execution in the current stack frame
  - Think of this like the visual studio function step over
- **until *location***
  - Continues until execution reaches the specified location or the current stack frame is exited
- **stepi/nexti**
  - Similar functionality to step and next but limited to machine instructions instead of source lines

GDB Basics - Stepping 2

- **skip *opts***
  - Prevent the stepping into certain functions
  - The value of opts can be found via help skip
- **skip function *linespec***
  - Prevents stepping into the function specified
- **skip file *filename***
  - Prevents stepping into any function in the file specified
- **info skip**
  - Lists the active skips
- **skip delete/enable/disable**
  - These work very similarly to breakpoints

GDB Basics - Examining Source

- **list *linenum***
  - Print lines centered around linenum in the current file, defaults to the current line number
- **list *function***
  - Print lines centered around the beginning of the function
- **list**
  - If the last lines were printed with a list command, it prints more lines
- **list-**
  - Prints lines before the lines previously listed
- **set listsize count**
  - Sets the default number of lines to list

GDB Basics - Working with Symbols

- **info symbol *symbol***
  - Displays some high level information about a symbol
- **whatis *symbol***
  - Displays the type of the symbol but does not unroll typedefs
- **ptype *symbol***
  - Displays the type of the symbol and unrolls typedefs to bare types
- **info macro *macro***
  - Displays the definition of a given macro
- **set *variable* = *expression***
  - Changes the value of a variable to equal the value of an expression (which can include the results of function calls)
- **p *expression***
  - Displays the current value of a given expression, usually a symbol name

GDB Basics - Displaying Expressions

- **display *expression***
  - Causes the value of an expression (usually a variable name) to be displayed on each execution step
  - The output format can be specified, see the help display in gdb for more information
- **undisplay *number***
  - Removes the display element with the specified number from the auto display list
- **info display**
  - Displays a table of the expressions that have been selected for automatic display

GDB Basics - Threads

- **info threads**
  - Displays a table of the currently executing threads
- **thread *n***
  - Switches to the thread with number specified
- **thread apply *n cmd***
  - Applies the command n to the thread numbered n
- **thread apply all *cmd***
  - Applies the command to all threads
- **thread name *name***
  - Sets the debugger name for the current thread (seen at info threads)
- **thread find *regex***
  - Displays a list of threads that contain the given regex in their name or id

GDB Basics - Managing Signals

- **info signals**
  - Displays a table of signals and the setting of the debugger with regard to stopping, passing them through to the target, and printing them
- **handle ...**
  - Instructs the debugger what actions to perform when a signal is sent to the process or raised by the process
  - Example:
    - handle SIGUSR2 nostop noprint
      - Instructs the DEBUGGER not to stop or print when SIGUSR2 is received by the process

GDB Basics - Examining Memory

- ***x/format address***
  - Format
    - Repeat count followed by a format letter and a size letter
    - Format letters are:
      - o(octal), x(hex), d(decimal), u(unsigned decimal), t(binary), f(float), a(address), i(instruction), c(char) and s(string)
    - Size letters are:
      - b(byte), h(halfword), w(word), g(giant, 8 bytes)
  - Address is a memory address

GDB Basics - TUI

- **To Toggle TUI mode**
  - CTRL-x a
- **Layout *name***
  - next/prev - move through the layouts
  - src - display the source and command windows
  - asm - display the assembly and command windows
  - split - display the source assembly and command windows
  - regs - when in src, display register, source and command window
  - regs - when in asm or split displays register, assembler, and command windows
- **Focus *name***
  - Changes which TUI window is active for scrolling
- **tui reg *group***
  - next/prev - move through the layouts
  - general, float, system, vector, all

GDB Basics - Disassembly 1

- **info line *location***
  - Prints the starting and ending addresses of the compiled code for the source line location
  - Default location is the current location
- **disass *args expression***
  - Dumps a range of memory as machine instructions
  - Arguments
    - /s causes the display to have mixed source and assembly
  - Expression is an address or function name
- **set disassembly-flavor *flavor***
  - Flavor is either att or intel
- **info registers**
  - Displays the values of certain registers
- **info all-registers**
  - Displays the values of all registers
- **info float**
  - Displays the state of the floating point unit
- **info vector**
  - Displays the state of the vector processing unit

Assembly and Calling Conventions

General Purpose Registers – Intel x86

Intel x86 architecture is based around a small set of general purpose registers

- %eax – Return values
- %edx – Dividend register (divide operations)
- %ecx – Count register (shift and string ops)
- %ebx – Stack frame pointer
- %esp – Stack Pointer
- %esi – Local register variable
- %edi – Local register variable

General Purpose Registers – Intel x64

The 32 bit registers are mapped to the lower 32 bits of the 64 bit registers

- **%rax – Return values**
- **%rdx – Dividend register (divide operations)**
- **%rcx – Count register (shift and string ops)**
- **%rbx – Stack frame pointer**
- **%rsp – Stack Pointer**
- **%rsi – Local register variable**
- **%rdi – Local register variable**

Floating point registers

The floating point unit on an Intel processor acts like a stack. Note that the floating point unit does not discriminate between float and double, all registers are more precise than double precision.

- %st(0) – common floating point return register
- %st(1) through %st(7) floating point stack values

Special Purpose Registers

There are a few special purpose registers that you will most likely not have to deal with directly.

- %eip / %rip – the instruction pointer
- %eflags / %rflags – the flags register

Example Assembly Instructions

Instruction	Meaning
movq %rax, %rbx	rbx = rax
movq $123, %rax	rax = 123
movq %rsi, -16(%rbp)	mem[rbp-16] = rsi
subq $10, %rbp	rbp = rbp -10
cmpl %eax %ebx	if eax == ebx, zero flag is set.
jmp <location>	unconditional jump
je <location>	jump to <location> if equal is set
jg,jge,jl,jle,jne,...	>,>=,<,<=,!=,...

Other Important Instructions

- **lea src, dest**
  - Load effective address
    - Calculates the address of the src opperand and loads it into the dest operand
- **call *name***
  - Performs a function call to name

Calling Conventions – cdecl (x86)

x86 implements several calling conventions. This slide will discuss the most common, cdecl

- **cdecl is the standard calling convention for c functions**
- **Arguments are pushed onto the stack in reverse order**
- **The stack is word aligned by convention**
- **%ebp, %ebx, %esi, %esp must be preserved by the function**

Calling Conventions – cdecl (x86)

Position	Contents	Frame	
4n+8(%ebp)	Argument word n	Previous	High Addresses
	...	Previous	
8(%ebp)	Argument word 0	Previous	
4(%ebp)	Return address	Current	
0(%ebp)	Previous %ebp(optional)	Current	
-4($ebp)	Unspecified	Current	
	...	Current	
0(%esp)	Variables	Current	Low Addresses

This is the typical stack layout for a function called using the cdecl calling convention on x86

Function Prologue – cdecl (x86)

The beginning of a typical function will look like this:

```
pushl %ebp       ; save frame pointer
movl %esp, %ebp  ; set new frame pointer
subl $80, $esp   ; make room for locals on the stack
pushl %edi       ; save local register
pushl %esi       ; save local register
pushl %ebx       ; save local register
```

Note that local stack variables are accessed directly from the stack, usually via offset from the stack frame pointer.

Function Epilogue – cdecl (x86)

The end of a typical function will look like this:

```
movl %edi, %eax    ; set up return variable
popl %ebx          ; restore ebx from stack
popl %esi          ; restore esi from stack
popl %edi          ; restore edi from stack
leave              ; restore frame pointer
ret                ; pop return address
```

Other Useful Tools

LDD

- **LLD is a tool that is used to diagnose when the dynamic loader (ld.so usually) is unable to resolve the runtime dependencies of a binary**
- **Usage:**
  - lld *binary*
- **When the dynamic library loader is unable to resolve a library file using the LD_LIBRARY_PATH environment variable and the optional rpath information stored in the ELF header in the binary, ldd will display an error for that library**
- **Ldd will also display the complete path to binaries that are being used to fulfill required libraries which can help to debug random crashes on program startup due to mismatching library versions**

nm

- **nm is a tool that lists information about symbols contained in a binary (either shared library or executable)**
- **Usage:**
  - nm *binary*
- **The output of nm is rather terse and the man page is a great reference**
- **The general output is in the form of address, segment, symbol name**
- **The segment is usually the more useful bit when using nm**
  - B and D are usually global variables
  - T is usually a function that is present within that binary
  - U is usually a function that is called from the binary but is not present within it e.g. it comes from an external library and is resolved at load time

objdump

- **Objdump is a tool that is used to dump object information out of a binary**
- **Objdump is also used to disassemble binaries for inspection without needing gdb**
- **Usage:**
  - objdump -T *binary*
    - Dumps out a list of symbols that the dynamic linker must resolve at run
  - Objdump -D *binary*
    - Dumps out the disassembly of the binary for inspection
  - Objdump -s *binary*
    - Dumps out the disassembly intermixed with the source code for inspection

xxd

- **xxd is a tool that is used to create hex dumps of a binary**
- **xxd is also capable of performing binary patching but this is unusual**
- **Think of xxd as a non-interactive hex editor**
- **Usage:**
  - xxd *binary*

strace

- **strace is a tool that is used to track system calls**
- **When a program runs and appears to hang up but it is difficult to debug for some reason (i.e. no symbols), you can still see what it is trying to do by using strace**
- **Usage:**
  - strace *binary*
- **Note that the strace man page provides a great deal more information that can render the output of strace much easier to read and understand**

License Information