

# Tracing Struct Accesses with Struct Stalker

A Journey into the Darkness of LLDB Scripting

Jeff Dileo

RECON MTL 2018



## LLDB Considered Harmful Painful

Jeff Dileo  
RECON MTL 2018



```
call_usermodehelper("/bin/sh", (char *[]){"/bin/sh", "-c", "whoami", NULL}, NULL, 5);
```

- @chaosdatumz
- I am a:
  - Unix aficionado
  - Consultant at NCC Group
- I like to do terrible things to/with/in:
  - programs
  - languages
  - runtimes
  - memory
  - kernels
  - packets
  - bytes
  - ...



*I have come here to chew bubblegum and criticize LLDB,  
and I'm all out of bubblegum.*

1. Background
2. Tool/demo
3. Introduction to LLDB and LLDB scripting
  - aka "grumbling about LLDB gotchas and how to script complicated things"
  - Other lessons from the trenches



It all starts with this, a fuzzer containing the ultimate problem.

- Was working on some libFuzzer stuff
  - LibFuzzer (LLVM project) is an "in-process, coverage-guided evolutionary fuzzing engine"
    - Link libFuzzer to code
    - LibFuzzer becomes the entry point
    - LibFuzzer entry point calls your fuzz entry point
    - You convert fuzz data input into arguments/structs/etc and pass to code to be tested
    - Primarily used for fuzzing "library" code
    - But what if you want to fuzz arbitrary application code?



It all starts with this, a fuzzer containing the ultimate problem.

- Was working on some libFuzzer stuff
  - LibFuzzer (LLVM project) is an "in-process, coverage-guided evolutionary fuzzing engine"
    - Link libFuzzer to code
    - LibFuzzer becomes the entry point
    - LibFuzzer entry point calls your fuzz entry point
    - You convert fuzz data input into arguments/structs/etc and pass to code to be tested
    - Primarily used for fuzzing "library" code
    - But what if you want to fuzz arbitrary application code?
- Problem: Which fields are actually used? needed?
  - Many complicated structs with a mix of internal and external structs/data



It all starts with this, a fuzzer containing the ultimate problem.

- Was working on some libFuzzer stuff
  - LibFuzzer (LLVM project) is an "in-process, coverage-guided evolutionary fuzzing engine"
    - Link libFuzzer to code
    - LibFuzzer becomes the entry point
    - LibFuzzer entry point calls your fuzz entry point
    - You convert fuzz data input into arguments/structs/etc and pass to code to be tested
    - Primarily used for fuzzing "library" code
    - But what if you want to fuzz arbitrary application code?
- Problem: Which fields are actually used? needed?
  - Many complicated structs with a mix of internal and external structs/data
- Idea: Profile field accesses



It all starts with this, a fuzzer containing the ultimate problem.

- Was working on some libFuzzer stuff
  - LibFuzzer (LLVM project) is an "in-process, coverage-guided evolutionary fuzzing engine"
    - Link libFuzzer to code
    - LibFuzzer becomes the entry point
    - LibFuzzer entry point calls your fuzz entry point
    - You convert fuzz data input into arguments/structs/etc and pass to code to be tested
    - Primarily used for fuzzing "library" code
    - But what if you want to fuzz arbitrary application code?
- Problem: Which fields are actually used? needed?
  - Many complicated structs with a mix of internal and external structs/data
- Idea: Profile field accesses
- Problem: GDB/LLDB variable/memory watchpoints were useless for this
  - Size and quantity limitations





- Pointer Sequence Reverser — <https://github.com/nccgroup/psr>
  - Written by former co-worker (Nicholas Collisson)
    - I helped a bit with the design/architecture though
  - Traced all executed instructions
  - Traced accesses of "interesting" addresses by setting restrictive page permissions and catching page faults
    - Using win32 debugging/memory management APIs (e.g. OpenProcess/VirtualAllocEx)
  - On trapped access, processes accessing instruction and backlog with Capstone
  - Identified source objects (origin objects referencing to the "interesting" memory)
    - By reversing pointer sequences of offsets and dereferences



- The opposite analysis (source to value) is possible using the same page fault technique
  - e.g. start from root object, lock its pages, then iterate for each pointer in it
    - Generally do the locking last as the debugger may fail to read the pointers if locked first
- Whipped up a poc that did it, and then later decided to try a proper rewrite
- Originally did this in LLDB, then used LLDB later for rewrite
  - Both experiences were traumatic



**And now for something  
completely different...**



- Struct Stalker is an LLDB Python script/command
  - For macosx and Linux
- Configured with multiple calls
  - Give it information on the function and variable to track
  - Eventually have it "start"
    - It will either start the process or continue
    - It will take control of the debugger until all tracked functions have returned (or the processes has terminated)
- Uses `mprotect(2)` to make pages containing target variables inaccessible
  - It can dereference pointers/references but also supports stack-based variables
    - This is needed as it starts tracking local variables before they are initialized
  - It iterates struct fields where possible to recursively lock pages
- Catches page faults to identify and track variable accesses



**DEMO**



- Primarily requires source to be useful
  - Was written to support libFuzzer-based flows
- Assumes:
  - Stack variable field accesses touch memory
  - Stack variables' locations are not reused within the same stack frame



<https://github.com/nccgroup/struct-stalker>  
(soon, needs some polish ;)



- Brian McFadden
- Nicholas Collisson






**...is that it?**



**...is that it?**

**no**



A stylized illustration featuring a large bear on the left and a bird on the right. Both animals are depicted with thick black outlines and are blowing beams of light that form a rainbow. The bear's beam is in the foreground, while the bird's beam is in the background. The background is a textured green and blue. A semi-transparent dark grey banner is overlaid across the center of the image, containing white text.

In Memoriam  
James "shenanigans" Shanahan (1982-2018)

**IT'S LLDB LESSON TIME!**



WHY???????????????????

- Originally to learn it
- It's sort of eating the world
  - Core platform is macosx/iOS, which requires insane mach APIs to debug
    - ptrace(2) implementation is a joke
    - par for the course for a platform that stores process metadata in very tiny fixed-size structs (kernel actually front truncates long paths [ $> 1023$ ] by segment [max 255] to store them)
    - Poor support from for other debuggers (getting gdb working is an effort now)
  - Riding the LLVM wave, it's being pulled into other platforms
    - Ubuntu packages (which doesn't even have proper packages for libc++)
- Supposedly very scriptable
  - but is it scriptable enough?



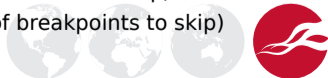
- LLDB is the LLVM debugger (major competitor to GDB)
- It appears designed more for machines (\*cough\* Xcode \*cough\*) than humans
- Built using LLVM components powering other tools like Clang (the C/C++ frontend)
- Directly supports source-based debugging of C, C++, Objective-C, and Swift
  - And anything with DWARF debug symbols
  - Also, of course, works for stripped binaries of unknown provenance
- Supported platforms (in order of support/feature status):
  - macosx/iOS
  - Linux
  - FreeBSD
  - Windows (WIP)
- Python API for scriptable debugging
  - Accessible from LLDB CLI process
  - Can also leverage LLDB components directly from python



- `$ lldb ...` (uses an `.lldbinit` file akin to `.gdbinit`)
- Lots of very verbose commands, with varying levels of GDB shorthand compatibility
- Free-form arguments are often placed after a `--` separator
- Decent documentation (`help` command followed by command, subcommand names)
  - Tab completion helps
- While many underlying things have IDs that print out with them, they are unused
  - Generally interacted with based on "#number" (index in a list)
- Most important command:
  - `settings set target.x86-disassembly-flavor intel`
- Most important alias (on macosx, because homebrew):
  - `alias lldb='PATH="/usr/bin:/bin:/usr/sbin:/sbin" lldb'`



- process
  - attach
  - connect (to remote debugger server)
  - launch (r <args...> works)
    - -X <true/false> (whether to shell expand args, enabled when using r alias)
    - -A <true/false> (whether to disable ASLR)
    - -i <file> (source STDIN from file)
    - -e <file> (redirect STDERR)
    - -o <file> (redirect STDOUT)
    - -s (stop at entry)  
Needed for address breakpoints. Otherwise, they *don't work* if set before process start/load.  
**Note:** Breaks before dynamically linked funcs/libs are resolved for expression.
  - handle (configure signal handling)
  - status (dumps process state, thread info (different format), disassemble -p)
  - continue (same as thread continue, but can specify number of breakpoints to skip)





- disassemble (x/<n>i <expr> still works)
  - -n <func\_name>
  - -f (current stack frame)
  - -p (from current instruction)
  - -c <num ins>
  - -b (show bytes)
  - -s <expr> (from address)
  - -a <expr> (whole function containing address)



- breakpoint (b <expr> mostly works, but d <expr> is short for disassemble)
  - set
  - list
  - command
    - -o <one-liner>
    - -s <lang> (command, python)
    - -F <python\_module.function>
    - -o <true/false> (one-shot)
  - delete



- thread
  - backtrace (prints thread call stack)
  - continue
    - [num...] (optionally supports a list of threads to continue)
  - list
  - info (dumps various pieces of information based on debug availability)
    - including the thread stop reason (e.g. step, breakpoint, page access error)
  - select (select "current" thread used for **f**rame commands)
  - jump (change program counter)
  - return (force return with value)
  - step-scripted (configure python to handle stepping)



- `frame`
  - `info` (print info about "current" stack frame)
    - This is a subset of the `thread info` output (which only covers the bottom most frame)
    - Output as part of `thread backtrace`
  - `select` (set the "current" stack frame)
    - `-r <offset>` (relative offset from "current" selected frame)
    - `<num>` (0 is the lowest frame)
  - `variable` (dump info about variables from "current" selected frame from debug info)
    - options enable filtering argument, local, global, static variables from each other



- memory
  - find (scan memory for byte sequence)
  - read (read data from process memory, lots of specific and useful options)
    - The GDB compatible `x/...` is still more useful for most cases
    - `-o <file>` (output to file, hexdump format by default)
      - Note:** This does not clear the original file, be careful if rerunning the same command.
    - `-b` (raw output)
  - write (write data to process memory)
    - `-i <file>` (source data from file)
    - `-o <N>` (offset N bytes into source file)



- `target` (manages selected debugging target, enables creating a new one)
  - `modules list` (dump memory mappings)
  - `modules dump` (dump various information from files, e.g. sections and symbols)
  - `modules lookup` (search for symbols, addresses, etc in loaded binary files)



- expression (evaluates a source expression in the process)
    - Generates code on the fly with a JIT and runs it
    - -- (void)puts("hello world")
    - -l <language> -- <code> (doesn't really work for C++ STL stuff)
    - -Z <N> (treat pointer as array of N elements)
    - -i <true/false> (ignore breakpoints hit while evaluating the expression)
    - <no expression> (drops to multi-line editor)
    - Some potential issues:
      - When using process launch -s, dynamically linked libraries haven't loaded. (On macosx, step until dyldbootstrap::start completes)
      - If stopped at the main breakpoint on a non-debug binary, expression fails. (Disable the breakpoint before using expression)
- Note:** While the error mentions a breakpoint was hit, -i true does not fix this.



## Get Swifty

- Normally, expressions are limited by what exists in the binary itself
  - Can't call C library functions if they aren't loaded
  - Can't evaluate Swift expressions for non-Swift binaries (the Swift runtime is missing)
- But you can load the needed code into the binary to allow such expressions

```
$ lldb -x test
(lldb) target create "test"
Current executable set to 'test' (x86_64).
(lldb) env DYLD_INSERT_LIBRARIES=/Applications/Xcode.app/Contents/Frameworks/libswiftCore.dylib
(lldb) b main
Breakpoint 1: where = test`main + 22 at test.cc:58, address = 0x0000000100000df6
(lldb) r
...
(lldb) expression -l swift -- print("aaaa" + "bbbb")
aaaabbbb
```





- Python 2.x on all platforms but Windows
- Very thin shim on top of C++
  - Many APIs take an output object as a parameter
  - Some objects have lifetimes, continuing to use them in python is UAF
  - Complicated threading model (more on this later)
  - Buggy as hell (can very easily segfault LLDB itself)
    - Maybe someone should try fuzzing debuggers from the attached process or remote side
  - Lots of magic numbers that are un-namespaced in Python and barely namespaced in C++
- The CLI commands are implemented using this and the C++ API
- Many APIs are redundant and invoke applicable calls on "linked" objects
  - Contain/provide references to objects that are not their direct parent/child
- Also, many classes are named very similarly



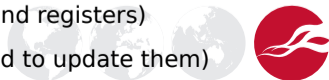
- Very poorly documented in general (LLDB is NOTORIOUS for this)
  - Sparse and, when available, often restates what the very long class/function name says
  - Often misleading or doesn't tell you how you are actually supposed to get them
  - Functions/fields may yield strings of numbers in base 10 or hex (never documented)
    - ...or numerics (rarely documented)



- "Public" classes in the `lldb` C++ namespace are generally prefixed with "SB"
  - Often avoid exposing otherwise useful APIs consumed internally by LLDB components
    - Generally through tons of `friend` class statements
    - It is unclear why these APIs are hidden from external code in the first place
- The "private" implementations are generally in the `lldb_private` C++ namespace
  - Often have a ton of convenience alias functions (for lazy devs) that take "SB" objects



- SBBreakpoint (breakpoint creation/configuration)
- SBDebugger (main interface for code)
  - Scripts always have access to one through args or module `lldb.debugger` variable
  - Generally used to access everything else (debugger->target->process->thread->frame)
- SBTarget (target-related functionality)
  - Provides many useful APIs for building one's own debugger out of the LLDB API
  - Provides a handle on:
    - Process
    - Platform (API for manipulating the host of the process; launch/attach/file IO/shell cmds/etc)
  - Many of its APIs are implemented by these underlying objects
- SBProcess (process-related functionality, also an iterable list of threads)
- SBThread (thread-related functionality, also an iterable list of frames)
- SBFrame (frame-related functionality, used to access variables and registers)
- SBValue (value of a variable, register, or expression; can be used to update them)



- SBStream (object used for collecting string output from function calls/objects)
- SBError (object used for collecting errors from function calls)
- SBBroadcaster (source of events)
  - Creating one is useless
  - Generally need to get a handle on an existing one from somewhere else
    - Many other objects emit events through their broadcaster
- SBListener (event listener you create and register to a broadcaster)
- SBEvent (event "data," a thin wrapper around a number without any real context)
- SBCommandInterpreter (for when you give up on the API and just run CLI commands)
- SBCommandReturnObject (get/set return value of CLI command)



- `command script import path/to/script.py`

```
import lldb
```

```
def __lldb_init_module(debugger, internal_dict):  
    lldb.command("my-cmd")(my_cmd_command_handler)
```

```
def my_cmd_command_handler(dbg, cmdline, res, idict):  
    # dbg: lldb.SBDebugger  
    # cmdline: single string of the command  
    # res: lldb.SBCommandReturnObject  
    # idict: dict of completely useless stuff otherwise obtainable from the lldb module  
    pass
```

- Protip: Parse the command string (above, `cmdline`) using an argument parser
  - e.g. `argparse`



- Breakpoints are created through the target object

```
def my_cmd_command_handler(dbg, cmdline, res, idict):  
    target = dbg.GetSelectedTarget()  
    bp = target.BreakpointCreateByName("main")  
    # bp = target.BreakpointCreateByAddress(...) # int  
    # bp.SetOneShot(True)  
    bp.SetScriptCallbackFunction('<filename>.bp_callback')  
  
def main_bp_callback(frame, bp_loc, idict):  
    # frame: lldb.SBFrame  
    # bp_loc: lldb.SBBreakpointLocation  
    # idict: same useless shared dict  
    pass
```

- The function must be specified by <filename>.<function\_name> as a string
- The SBFrame has a lifetime and should not be used after the function has returned



- The callback implementation is magic
  - The frame argument only lives until it returns
    - Stashing it and using it later can crash LLDB (use-after-free, double free, ...)
  - It runs concurrently with other Python code in the same interpreter
- Magic is not good for scriptable debugging
  - Especially significant non-determinism
  - LLDB releases/obtains the Python GIL on entering/leaving `lldb` FFI code
  - Means that callback code and background code essentially run concurrently
    - more on "background" code later
  - CPython itself is barely avoiding memory corruption...
    - ...but the `lldb` module and breakpoint callbacks are *not* threadsafe
- Very easy to crash LLDB (memory corruption, ...)





- Often, it is simpler/easier to run CLI commands than write baroque and esoteric code
  - Protip: Where possible, prototype with CLI commands, then replace with API calls

```
def main_bp_callback(frame, bp_loc, idict):  
    lldbrepl = lldb.debugger.GetCommandInterpreter()  
    pszres = lldb.SBCommandReturnObject()  
  
    lldbrepl.HandleCommand('expression --flat -- (size_t)sizeof(void*)', pszres)  
  
    POINTER_SIZE = int(pszres.GetOutput().split('=')[1].strip(), 10)
```



- expression is simple enough to API-ify
  - `SBTarget.EvaluateExpression(...)` takes a `SBExpressionOptions`
    - Just configures the code settings, not output options like `--flat`
  - Output options use a (C++) `DumpValueObjectOptions` data formatter internally
  - `EvaluateExpression` returns an `SBValue`
    - Often better than string parsing

```
def main_bp_callback(frame, bp_loc, idict):  
    target = lldb.debugger.GetSelectedTarget()  
  
    val = target.EvaluateExpression('(size_t)sizeof(void*)')  
  
    POINTER_SIZE = val.GetValue()
```



## Intermediate Scripting — Guard Pages

- On macosx/Linux, `mprotect(2)`
- However, LLDB does not gracefully handle when the stack becomes inaccessible
  - Gets stuck in `mprotect(2)` right after the syscall if stack frame permissions are modified
  - Existing frame objects become permanently invalidated in this situation
    - Including the `lldb.frame` module global and callback frame parameter

```
def mprotect(addr, len, flags, res, consts): # shift sp down a page before mprotect(2)
    target = lldb.debugger.GetSelectedTarget()
    thread = target.GetProcess().GetSelectedThread()
    frame = thread.GetSelectedFrame()

    orig_stack_str = frame.FindRegister("rsp").GetValue()
    new_stack_str = hex(int(orig_stack_str, 16) - consts['PAGE_SIZE'])
    frame.FindRegister("rsp").SetValueFromCString(new_stack_str)
    target.EvaluateExpression('(int)mprotect(0x{:x}, {}, {})'.format(addr, len, flags))

    thread = target.GetProcess().GetSelectedThread()
    frame = thread.GetSelectedFrame() # critically important to have this line
    frame.FindRegister("rsp").SetValueFromCString(orig_stack_str)
```



- LLDB only supports callbacks for breakpoints
  - Page fault-based process stops are not breakpoints
- We need to directly control all execution to handle when the debugger stops
- There aren't many examples of such code in the wild
  - Pretty much just LLDB uses its own libraries
    - Also, the unit test code is misleading since they exist to test behaviors, not work



- Broadcasters emit events to registered listeners
  - Typically, "broadcasters" (or things that "have" them) are complex components that maintain/move between states
  - When their state changes, they emit an event to listeners registered to them
  - There isn't much point in scripts creating broadcasters, but libraries may benefit to do so
- The LLDB C++ codebase defines the following kinds of event broadcasters:
  - "lldb.target" • "lldb.thread"
  - "lldb.targetList"
  - "lldb.process" • "lldb.commandInterpreter"
  - "lldb.communication" (used with remote debug servers)
  - "lldb.anonymous" (default broadcaster name)
- None of these resemble the word "debugger"



- Broadcasters emit events to registered listeners
  - Typically, "broadcasters" (or things that "have" them) are complex components that maintain/move between states
  - When their state changes, they emit an event to listeners registered to them
  - There isn't much point in scripts creating broadcasters, but libraries may benefit to do so
- The LLDB C++ codebase defines the following kinds of event broadcasters:
  - "lldb.target" • "lldb.thread"
  - "lldb.targetList"
  - "lldb.process" • "lldb.commandInterpreter"
  - "lldb.communication" (used with remote debug servers)
  - "lldb.anonymous" (default broadcaster name)
- None of these resemble the word "debugger"
  - tools/driver/Driver.\* (the lldb binary's main loop) actually subclasses SBBroadcaster
  - It is just a thin shim that creates and starts an SBDebugger
  - Each SBDebugger object has its own SBCommandInterpreter
    - This SBCommandInterpreter object runs the entire CLI UI



- SBListeners are “wait-able” filter objects that can be registered to broadcasters
  - Event type bitmasks checks are actually implemented in the broadcaster
- An associated broadcasters “peek” events into the listener (the terminology in the LLDB codebase is just as bad as its documentation)
  - If waiting, the listener’s `Get(Next)Event*()/WaitForEvent()` will return with the event
    - `WaitForEvent()` takes a timeout value
- `source/Core/Debugger.cpp` (private SBDebugger implementation) has an event loop
  1. Registers its listener to process, thread, command interpreter
  2. Loops on listener’s `GetEvent()`
  3. Compares event’s broadcaster by class or directly against its command interpreter
  4. Handles the event type value accordingly for each



- SBListeners are “wait-able” filter objects that can be registered to broadcasters
  - Event type bitmasks checks are actually implemented in the broadcaster
- An associated broadcasters “peek” events into the listener (the terminology in the LLDB codebase is just as bad as its documentation)
  - If waiting, the listener’s `Get(Next)Event*()/WaitForEvent()` will return with the event
    - `WaitForEvent()` takes a timeout value
- `source/Core/Debugger.cpp` (private SBDebugger implementation) has an event loop
  1. Registers its listener to process, thread, command interpreter
  2. Loops on listener’s `GetEvent()`
  3. Compares event’s broadcaster by class or directly against its command interpreter
  4. Handles the event type value accordingly for each

*[G]ood artists copy, great artists steal.*





- SBListeners are “wait-able” filter objects that can be registered to broadcasters
  - Event type bitmasks checks are actually implemented in the broadcaster
- An associated broadcasters “peek” events into the listener (the terminology in the LLDB codebase is just as bad as its documentation)
  - If waiting, the listener’s `Get(Next)Event*()/WaitForEvent()` will return with the event
    - `WaitForEvent()` takes a timeout value
- `source/Core/Debugger.cpp` (private SBDebugger implementation) has an event loop
  1. Registers its listener to process, thread, command interpreter
  2. Loops on listener’s `GetEvent()`
  3. Compares event’s broadcaster by class or directly against its command interpreter
  4. Handles the event type value accordingly for each

*[G]ood artists copy, great artists steal.  
And we have, you know, always been shameless  
about stealing great ideas.*

-Steve Jobs



- Simple enough to implement, just need to know all event values for each broadcaster
  - Undocumented, but I've included the entire list for each at the end of the slides

```
def debugger_loop(dbg):  
    dbg.SetAsync(True)  
    listener = lldb.SBListener('my_debugger_loop')  
    process = dbg.GetSelectedTarget().GetProcess()  
    process.GetBroadcaster().AddListener(listener, 0xffffffff)
```

...

- Create a listener object and register it to the process broadcaster



```
...
while True:
    if dbg.GetSelectedTarget().GetProcess().GetState() == lldb.eStateExited:
        # shortcut to prevent waiting for an event that won't arrive
        break
    try:
        event = lldb.SBEvent()
        #res = listener.WaitForEvent(4294967295, event) # UINT32_MAX # misses exit
        res = listener.WaitForEvent(1, event)
        if not res:
            continue
        if not lldb.SBProcess.EventIsProcessEvent(event):
            continue
    ...
```

- Loop forever, waiting for an event
- Make sure the event is from a broadcaster you expect



```
...
if event.GetType() != lldb.SBProcess.eBroadcastBitStateChanged:
    continue
state = lldb.SBProcess.GetStateFromEvent(event)
if state == lldb.eStateInvalid || state == lldb.eStateRunning:
    continue
elif state == lldb.eStateStopped:
    ### DO STUFF HERE ###
    process.Continue()
else:
    process.Continue()
except Exception:
    traceback.print_exc()
break
```

- Manually filter and handle the events accordingly
  - Could have used a more fine-grained event mask



- Run into issues if breakpoint callbacks are used
  - Fix: Implement our own breakpoint callbacks
    - Can check if the process stop reason is "breakpoint" or "one-shot breakpoint"
- Different platforms have different process state description string formats
  - ...and different reasons for the same occurrences (e.g. page faults)
    - EXC\_BAD\_ACCESS for macosx
    - SIGSEGV for Linux
  - Fix: Parse description string based on platform
- Need to have LLDB catch page fault signals (different on each platform)
  - Fix: The process handle command:
    - `process handle SIGBUS --notify true --pass false --stop true (macosx)`
    - `process handle SIGSEGV --notify true --pass false --stop true (Linux)`



- The lesson is over.
- You are now LLDB scripting wizards. \*poof\*

## Questions?

jeff.dileo@nccgroup.trust

@chaosdatumz



## North America

Atlanta  
Austin  
Chicago  
New York  
San Francisco  
Seattle  
Sunnyvale  
Waterloo, Ontario

## Europe

Manchester - Head Office  
Amsterdam  
Basingstoke  
Cambridge  
Copenhagen  
Cheltenham  
Edinburgh  
Glasgow  
Leatherhead  
Leeds  
London  
Luxembourg  
Malmö  
Milton Keynes  
Munich  
Vilnius  
Wetherby  
Zurich

## Australia

Sydney



# Tracing Struct Accesses with Struct Stalker

A Journey into the Darkness of LLDB Scripting

Jeff Dileo

RECON MTL 2018





- The next several slides provide some useful references
  - Primarily the undocumented magic enums needed to do anything useful with LLDB



- Target (include/lldb/Target/Target.h)
  - lldb.SBTarget.eBroadcastBitBreakpointChanged
  - lldb.SBTarget.eBroadcastBitModulesLoaded
  - lldb.SBTarget.eBroadcastBitModulesUnloaded
  - lldb.SBTarget.eBroadcastBitWatchpointChanged
  - lldb.SBTarget.eBroadcastBitSymbolsLoaded
- TargetList (include/lldb/Target/TargetList.h)
  - eBroadcastBitInterrupt = (1 << 0) (not accessible from Python)



- Process (`include/lldb/Target/Process.h`)
  - `lldb.SBProcess.eBroadcastBitStateChanged`
  - `lldb.SBProcess.eBroadcastBitInterrupt`
  - `lldb.SBProcess.eBroadcastBitSTDOUT`
  - `lldb.SBProcess.eBroadcastBitSTDERR`
  - `lldb.SBProcess.eBroadcastBitProfileData`
  - `lldb.SBProcess.eBroadcastBitStructuredData`



- Thread (`include/lldb/Target/Thread.h`)
  - `lldb.SBThread.eBroadcastBitStackChanged`
  - `lldb.SBThread.eBroadcastBitThreadSuspended`
  - `lldb.SBThread.eBroadcastBitThreadResumed`
  - `lldb.SBThread.eBroadcastBitSelectedFrameChanged`
  - `lldb.SBThread.eBroadcastBitThreadSelected`



- Command Interpreter (`include/lldb/Interpreter/CommandInterpreter.h`)
  - `lldb.SBCommandInterpreter.eBroadcastBitThreadShouldExit`
  - `lldb.SBCommandInterpreter.eBroadcastBitResetPrompt`
  - `lldb.SBCommandInterpreter.eBroadcastBitQuitCommandReceived`
  - `lldb.SBCommandInterpreter.eBroadcastBitAsynchronousOutputData`
  - `lldb.SBCommandInterpreter.eBroadcastBitAsynchronousErrorData`



- `Communication(include/lldb/Core/Communication.h)`
  - `lldb.SBCommunication.eBroadcastBitDisconnected`
  - `lldb.SBCommunication.eBroadcastBitReadThreadGotBytes`
  - `lldb.SBCommunication.eBroadcastBitReadThreadDidExit`
  - `lldb.SBCommunication.eBroadcastBitReadThreadShouldExit`
  - `lldb.SBCommunication.eBroadcastBitPacketAvailable`
  - `lldb.SBCommunication.eBroadcastBitNoMorePendingInput`



- `include/lldb/lldb-enumerations.h`
  - `lldb.eStateInvalid`
  - `lldb.eStateUnloaded` (Process valid, but not loaded)
  - `lldb.eStateConnected` (Process connected to debug server, but not launched/attached)
  - `lldb.eStateAttaching` (Process it trying to attach)
  - `lldb.eStateLaunching` (Process is currently launching)
  - `lldb.eStateStopped` (Process/Thread is stopped/paused)
  - `lldb.eStateStepping` (Process/Thread is running)
  - `lldb.eStateStepping` (Process/Thread is currently stepping)
  - `lldb.eStateCrashed` (Process/Thread has crashed)
  - `lldb.eStateDetached` (Process has detached)
  - `lldb.eStateExited` (Process has exited)
  - `lldb.eStateSuspended` (Process/Thread is suspended so other procs/threads can run)



- `include/lldb/lldb-enumerations.h`
  - `lldb.eStopReasonInvalid`
  - `lldb.eStopReasonNone`
  - `lldb.eStopReasonTrace`
  - `lldb.eStopReasonBreakpoint`
  - `lldb.eStopReasonWatchpoint`
  - `lldb.eStopReasonSignal`
  - `lldb.eStopReasonException`
  - `lldb.eStopReasonExec (re-exec'd)`
  - `lldb.eStopReasonPlanComplete`
  - `lldb.eStopReasonThreadExiting`
  - `lldb.eStopReasonInstrumentation`





Most other stuff directly within the lldb module (i.e. lacking namespacing)

- See `include/lldb/lldb-enumerations.h`
  - Launch flags
  - Thread run modes
  - Byte orderings
  - Encodings
  - Display format definitions
  - Description levels for `GetDescription/str()`
  - Script interpreter types
  - Command return statuses
  - Expression results
  - Connections status types
  - Error types
  - Value types
  - ... (So many things!)



- LLDB doesn't support following the child process after fork(2)
  - (gdb) set follow-fork-mode child
- So I hacked up one on request
  - (lldb) follow-child

<https://github.com/chaosdata/lldb-misc/tree/master/follow-child>

(slide added post-talk)



# Tracing Struct Accesses with Struct Stalker

A Journey into the Darkness of LLDB Scripting

Jeff Dileo

RECON MTL 2018

