LLDB

Apple’s powerful debugging tool

Outline

- History of LLDB
- What is LLDB?
- Solving problems in our code
- Why use LLDB?
- Getting started with LLDB
- LLDB commands and Regular Expressions
- Use cases

Outline (cont.)

- Creating custom commands
- Persisting commands
- Integrating Chisel from Facebook (Install and usage)
- Creating custom Python scripts
- Swift REPL
- Assembly basics
- Honourable Mentions
- Peeping under the hood of News on the Simulator

History

- Apple’s older debugger - GDB (GNU Debugger)
- LLDB was built as part of the LLVM project
- LLDB is built upon LLVM standard libraries and uses Clang as the expression parser so it’s upto date whenever the compiler is updated

LLDB(Low-level debugger)

- LLDB is Xcode's native debugger or more technically a system debugger library
- It is the modern replacement for GDB
- Attaches to your current process in Xcode when you hit a breakpoint
- Contains a python module which enables you to have personalised scripts
- Blazingly fast and get's the job done

Approaching a problem

- What are the different ways in which we diagnose a problem in our source code?
- Console output for a crash or NSLog
- Breakpoint GUI (Gutter). Expand on types of breakpoints in the GUI
- View Debugger (Inspect the view hierarchy)
- Static analysers + Instruments
- Finally, LLDB commands ‘p’ and ‘po’

Why delve deeper?

- Speeds up your workflow dramatically
- Very powerful when used to it's full extent
- You can tackle some fairly complex problems without editing the source
- Explore your run time environment
- Automate repetitive debug tasks using python scripts
- Breaks through the limitations of other debugging workflows

LLDB command syntax

object action [options] [arguments]

Example

breakpoint set —name main

LLDB Basics

- How do i start to use it?
- Help and apropos
- p & po
- expression
- Debugger, Target, process, thread and frame
- image lookup (target modules)
- breakpoint
- b
- rbreak
- Commands resolve themselves to the nearest match

Using basic commands

- p & po in action
- Note : The expression command modifies your code using the LLVM JIT
- expression command can switch between JITs

Basic use cases

- Alter arguments to a function call during runtime using expression
- ‘l’ command for going through source code
- Thread until command. Useful for debugging flow

Explore source code within the debugger

l [line-num]

You can list source code around a particular address!

l [address]

Adding breakpoints

There are 3 main commands for adding breakpoints through LLDB

- breakpoint
- b
- rbreak

Examples

breakpoint delete <breakpoint num>

b ViewController.swift:20

rbreak viewDid -s <module name>

Adding actions to breakpoints

breakpoint command add <cmd-options> [<breakpt-id>]

```
(lldb) br command add 4.2
Enter your debugger command(s).  Type 'DONE' to end.
> po @"I hit this breakpoint! But i will not stop"
> continue
> DONE
```

Specifying continue as the last subroutine will prevent the breakpoint from stopping the process!

You can also add python commands here directly using the '-s option'

```
(lldb) br command add -s python 4.2
Enter your Python command(s). Type 'DONE' to end.
def function (frame, bp_loc, internal_dict):
    """frame: the lldb.SBFrame for the location at which you stopped
       bp_loc: an lldb.SBBreakpointLocation for the breakpoint location information
       internal_dict: an LLDB support object not to be used"""
    print "hello world"
    DONE
```

Adding conditions to breakpoints

Syntax - breakpoint modify -c ‘<condition>’ [<breakpt-id>]

```
(lldb) b -[SecondViewController upgradeCounter]
Breakpoint 2: where = SDLLDBObjectiveC`-[SecondViewController upgradeCounter] at SecondViewController.m:28, address = 0x000000010cffc4d0
(lldb) br modify -c '_counter.integerValue > 6' 2
```

Note - Adding another condition will replace the previous condition

Image lookup. Its great for searching through code!

- image lookup goes through the symbols in your runtime environment
- You can essentially find anything. Anything you want if you know how to look for it.
- Once you have it. You can execute it. Private frameworks included (objective C is much more lenient than swift in this regard)
- Attacking singletons can help reverse engineer code you don’t have the source for.
- image list can be used to print out all the modules currently within the runtime environment

Crash course in Regex

- Why regex? Regex is the key to a faster debugging workflow. Bypass writing complete function signatures and do case insensitive search! ViewDidLoad function signature in swift - ‘SDLLDB.ViewController.viewDidLoad () -> ()’
- Regex option for image lookup ‘image lookup -r’
- ‘.*’ is used as a prefix and suffix to your regex input by default
- Two ways to put spaces in regex “\ ” or \s
- Case insensitive search (?'i)
- ‘^’ denotes the start of the regex
- ‘$’ denotes the end of the regex
- ‘.’ denotes any character

Some interesting use cases that we can solve using what we've learned

- I want to break on a particular setter which is autosynthesized by the compiler without adding it to the source and using a GUI breakpoint. Heck i want all of them implemented by a certain class.
- Put a breakpoint on the setter and getter using a single command.

A Few more use cases

- I want to break on every single viewDidLoad. Okay maybe restrict the viewDidLoads only in my app ?
- Seeing a function in a crash log you can’t find? Image lookup!
- Image lookup is great for searching for functions Apple uses to debug their own code. The private instance method ‘_methoddescription’ on the NSObject class is a good example of this.

Have a function that gets called only in a certain state?

Put a breakpoint and fire it right from LLDB with the don’t ignore breakpoints option

expression -i false - - <call-that-function-here>

How do you put one-shot breakpoints or silent breakpoints which log to the console ?

breakpoint set -o -f ViewController.swift -l 14

br command add

Going even further

- Find every method in UIKit

  image lookup -rn . UIKit

- Put a breakpoint on every single function of a class to see how it works

  rbreak <class-name> -s <module-name>

- Have a new framework that you’re including and want to see how it works? Put a breakpoint on every function in it!

  rbreak . -s <framework-name>

Hey can we shorten the most used commands to make using this even faster?

This is where ‘command [alias|regex]’ comes in

Syntax

command alias <cmd-options> -- <alias-name> <cmd-name> [<options-for-aliased-command>]

command regex f ’s/(<lovely-regex-goes-here>)/<command-goes-here>/‘

Examples

command alias bp breakpoint

command alias bpl breakpoint list

command regex mylookup ‘/s(.+)/image lookup -rn %1/’

Find methods listed in a protocol?

```
(lldb) po objc_getProtocol("UITableViewDelegate")
<Protocol: 0x107ce9870>

(lldb) p protocol_copyMethodDescriptionList((Protocol *)0x107ce9870, YES, YES, 0)
(struct objc_method_description *) $35 = 0x0000000000000000
(lldb) p protocol_copyMethodDescriptionList((Protocol *)0x107ce9870, NO, YES, 0)
(struct objc_method_description *) $36 = 0x00007fe4a3c11680
(lldb) po NSStringFromSelector($36[0].name)
tableView:willDisplayCell:forRowAtIndexPath:

(lldb) |
```

You can put all this logic into a multiline expression and make it an alias!

command regex pprotocol 's/(.+)/expression -lobjc -O -- <insert-multiline-logic-here>/'

Command regex wins all

```
(lldb) pprotocol UITableViewDelegate

Protocol: UITableViewDelegate, <Protocol: 0x107ce9870>
========================================
 (Required)
 (Optional)
- tableView:willDisplayCell:forRowAtIndexPath:, v40@0:8@16@24@32
- tableView:willDisplayHeaderView:forSection:, v40@0:8@16@24q32
- tableView:willDisplayFooterView:forSection:, v40@0:8@16@24q32
- tableView:didEndDisplayingCell:forRowAtIndexPath:, v40@0:8@16@24@32
- tableView:didEndDisplayingHeaderView:forSection:, v40@0:8@16@24q32
- tableView:didEndDisplayingFooterView:forSection:, v40@0:8@16@24q32
- tableView:heightForRowAtIndexPath:, d32@0:8@16@24
- tableView:heightForHeaderInSection:, d32@0:8@16q24
- tableView:heightForFooterInSection:, d32@0:8@16q24
- tableView:estimatedHeightForRowAtIndexPath:, d32@0:8@16@24
- tableView:estimatedHeightForHeaderInSection:, d32@0:8@16q24
- tableView:estimatedHeightForFooterInSection:, d32@0:8@16q24
- tableView:viewForHeaderInSection:, @32@0:8@16q24
- tableView:viewForFooterInSection:, @32@0:8@16q24
- tableView:accessoryTypeForRowWithIndexPath:, q32@0:8@16@24
- tableView:accessoryButtonTappedForRowWithIndexPath:, v32@0:8@16@24
- tableView:shouldHighlightRowAtIndexPath:, B32@0:8@16@24
- tableView:didHighlightRowAtIndexPath:, v32@0:8@16@24
- tableView:didUnhighlightRowAtIndexPath:, v32@0:8@16@24
- tableView:willSelectRowAtIndexPath:, @32@0:8@16@24
- tableView:willDeselectRowAtIndexPath:, @32@0:8@16@24
- tableView:didSelectRowAtIndexPath:, v32@0:8@16@24
- tableView:didDeselectRowAtIndexPath:, v32@0:8@16@24
- tableView:editingStyleForRowAtIndexPath:, q32@0:8@16@24
- tableView:titleForDeleteConfirmationButtonForRowAtIndexPath:, @32@0:8@16@24
- tableView:editActionsForRowAtIndexPath:, @32@0:8@16@24
- tableView:shouldIndentWhileEditingRowAtIndexPath:, B32@0:8@16@24
- tableView:willBeginEditingRowAtIndexPath:, v32@0:8@16@24
- tableView:didEndEditingRowAtIndexPath:, v32@0:8@16@24
- tableView:targetIndexPathForMoveFromRowAtIndexPath:toProposedIndexPath:, @40@0:8@16@24@32
- tableView:indentationLevelForRowAtIndexPath:, q32@0:8@16@24
- tableView:shouldShowMenuForRowAtIndexPath:, B32@0:8@16@24
- tableView:canPerformAction:forRowAtIndexPath:withSender:, B48@0:8@16:24@32@40
- tableView:performAction:forRowAtIndexPath:withSender:, v48@0:8@16:24@32@40
- tableView:canFocusRowAtIndexPath:, B32@0:8@16@24
- tableView:shouldUpdateFocusInContext:, B32@0:8@16@24
- tableView:didUpdateFocusInContext:withAnimationCoordinator:, v40@0:8@16@24@32
- indexPathForPreferredFocusedViewInTableView:, @24@0:8@16
```

Persisting commands

- User defined commands only last for that particular debug session instance. The next time Xcode launches, they will not be available

- LLDB initialisation file

- LLDB looks for this file at the path ‘~/.lldbinit-Xcode’, specifically for Xcode. If it can’t find this, it will default to ‘~/.lldbinit’ and if it can’t find that either then it will skip this step.

- How do we go about creating lldbinit?

- We can put aliases, LLDB settings and python script imports here and they will get loaded into Xcode every time we attach our process to LLDB!

Integrating Facebook's chisel

- https://github.com/facebook/chisel
- Facebook's chisel is an incredibly powerful set of python scripts
- To install, clone the repo and add this line to your .lldbinit file - '**command script import <path-to-chisel-directory>/fbllldb.py**'
- bmessage can put a breakpoint on a function in the superclass even if the subclass doesn't override that method
- Designer asking you to edit something in front of him? Chisel to the rescue! Use commands like visualise, pviews, caflush and pvc to quickly iterate changes.

.lldbinit-Xcode

```
1  command script import /Users/Hazmat/chisel/fbllldb.py
2
```

Creating Custom Python Scripts

- Create our python file (eg - aamir.py)
- Add '**command script import <full-path-to-your-script>**' to your .lldbinit file
- Add magic source code to your python file
- Your function will now appear in the lldb help listing
- You can also debug your own python scripts using the '-g' option

.lldbinit-Xcode

```
1  command script import /Users/Hazmat/chisel/fblldb.py
2  command script import /Users/Hazmat/Downloads/aamir.py
```

Magic python source code

```python
import lldb

def __lldb_init_module(debugger, internal_dict):
    debugger.HandleCommand('command script add -f getAppDelegate.getAppDelegateFunction getAD')

def getAppDelegateFunction(debugger, command, result, internal_dict):
    debugger.HandleCommand('expression -lobjc -g -- [(UIApplication *)[UIApplication sharedApplication] delegate]')
```

- **__lldb_init_module** - Called during initialisation of the debugger

- **getAppDelegateFunction** - Our custom function based on the function prototype definition for python functions (given below). The HandleCommand function executes the given expression in the lldb debugger instance

Python function prototype

def MyCommand_Impl(debugger, user_input, result, unused)

Swift REPL

- Allows you to add code to an already existing debug session
- You can prototype new code on top of your existing code base

```
(lldb) repl
9> func printName() {
10. print("Aamir")
11. }
12> printName()
Aamir
13>
```

When can this be helpful?

- Add extensions to existing protocols to augment functionality on the fly
- Ad-hoc testing

Registers and the Assembly

- Two types of architectures - x86(macOS) and ARM(iOS)
- Two flavours of assembly - Intel vs AT&T
- Assembly format - **operand destination source** (**xor rdi rdi**)

Register Calling Conventions

First parameter - $rdi

Second parameter - $rsi

Third parameter - $rdx

Fourth parameter - $rcx

Return value - $rax

Implement a symbolic breakpoint to log every viewDidAppear using registers

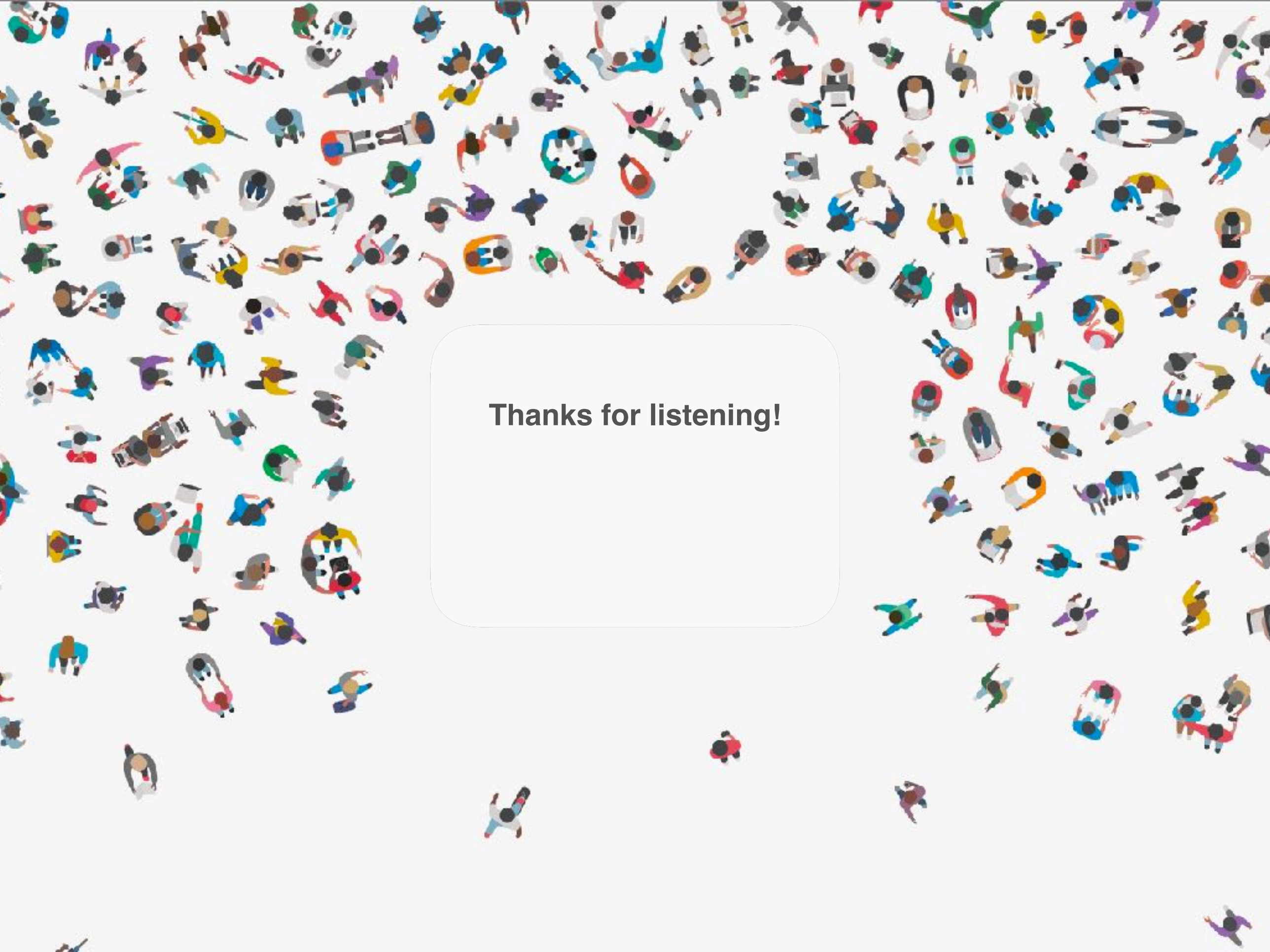
objc_msgSend(rdi,rsi,rdx)

```
(lldb) br list
Current breakpoints:
6: name = '-[UIViewController viewDidAppear:]', locations = 1, resolved = 1, hit count = 0
  6.1: where = UIKit`-[UIViewController viewDidAppear:], address = 0x000000010b0c53d6, resolved, hit count = 0

(lldb) br command add 6
Enter your debugger command(s).  Type 'DONE' to end.
> po $rdi
> DONE
 po $rdi
<SecondViewController: 0x7fa492f06c50>
```

Honorable mentions

- Watch points and their limitations
- LLDB is also available via the terminal
- Custom Data formatters
- https://github.com/neonichu/trolldrop
- 413 - Advanced Debugging WWDC2013
- Advanced Apple Debugging & Reverse Engineering - Derek Selander



Thanks for listening!