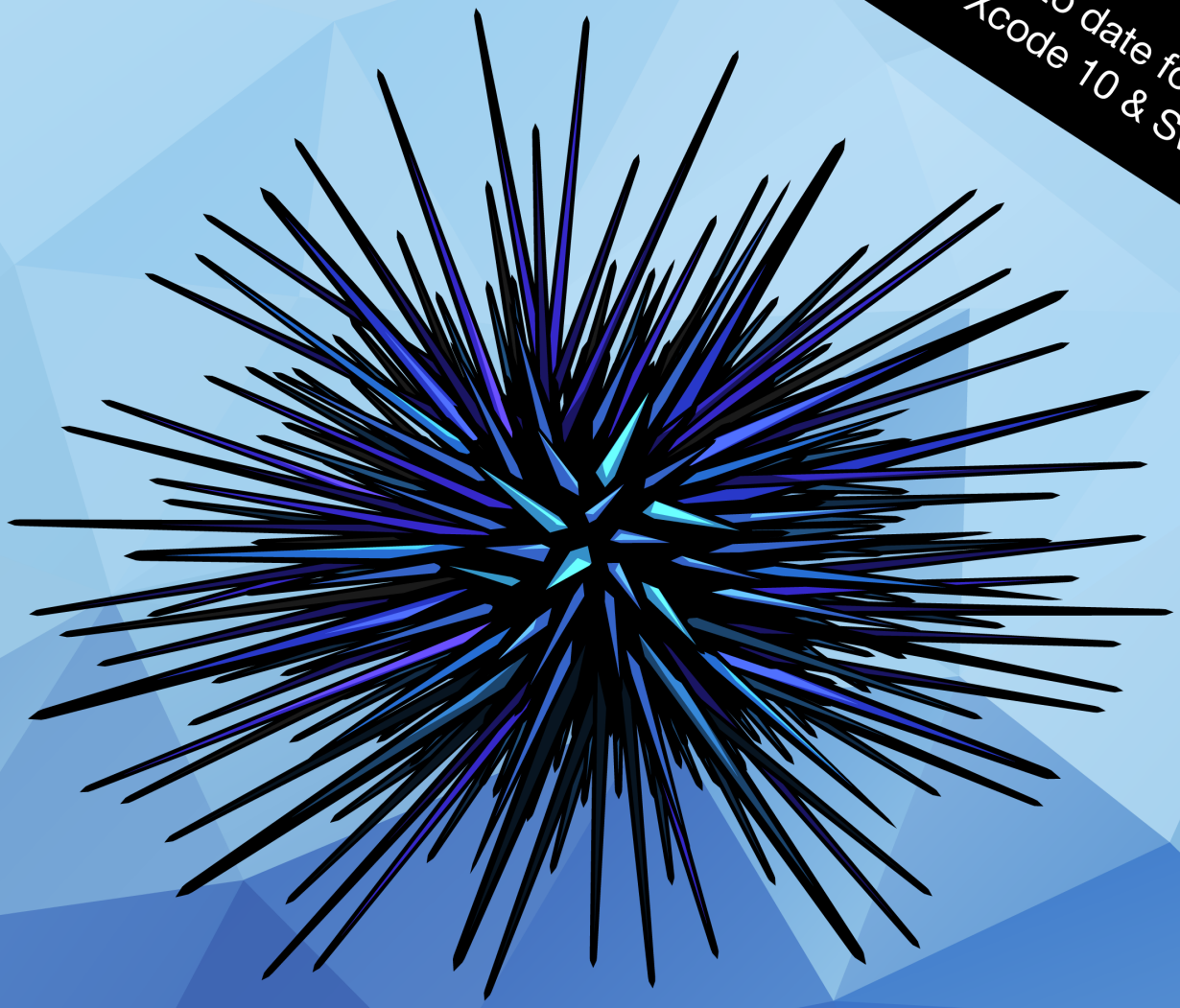
Up to date for iOS 12, Xcode 10 & Swift 4.2



Advanced Apple Debugging & Reverse Engineering

THIRD EDITION

Exploring Apple code through LLDB, Python, and DTrace

By Derek Selander

Advanced Apple Debugging & Reverse Engineering

Derek Selander

Copyright ©2018 Razeware LLC.

Notice of Rights

All rights reserved. No part of this book or corresponding materials (such as text, images, or source code) may be reproduced or distributed by any means without prior written permission of the copyright owner.

Notice of Liability

This book and all corresponding materials (such as source code) are provided on an “as is” basis, without warranty of any kind, express of implied, including but not limited to the warranties of merchantability, fitness for a particular purpose, and noninfringement. In no event shall the authors or copyright holders be liable for any claim, damages or other liability, whether in action or contract, tort or otherwise, arising from, out of or in connection with the software or the use of other dealing in the software.

Trademarks

All trademarks and registered trademarks appearing in this book are the property of their own respective owners.

Dedication

"I would like to thank my wife, Brittany, for all her love and support while I silently wept in the fetal position trying to get this book out the door."

— Derek Selander

About the Author



Derek Selander is the author of this book. His interest with debugging grew when he started exploring how to make (the now somewhat obsolete) Xcode plugins and iOS tweaks on his jailbroken phone, both of which required exploring and augmenting programs with no source available. In his free time, he enjoys pickup soccer, guitar, and playing with his two doggies, Jake & Squid.

About the Editors



Chris Belanger is the editor of this book. Chris is the Editor-in-Chief for raywenderlich.com. He was a developer for nearly 20 years in various fields from e-health to aerial surveillance to industrial controls. If there are words to wrangle or a paragraph to ponder, he's on the case. When he kicks back, you can usually find Chris with guitar in hand, looking for the nearest beach. Twitter: @crispytwit.



Matt Galloway is a software engineer with a passion for excellence. He stumbled into iOS programming when it first was a thing, and has never looked back. When not coding, he likes to brew his own beer.



Darren Ferguson is the final pass editor of this book. He is a Software Developer, with a passion for mobile development, for a leading systems integration provider based out of Northern Virginia in the D.C. metro area. When he's not coding, you can find him enjoying life with his wife and daughter trying to travel as much as possible.

About the Artist



Vicki Wenderlich is the designer and artist of the cover of this book. She is Ray's wife and business partner. She is a digital artist who creates illustrations, game art and a lot of other art or design work for the tutorials and books on raywenderlich.com. When she's not making art, she loves hiking, a good glass of wine and attempting to create the perfect cheese plate.

Table of Contents: Overview

Introduction 16

Section I: Beginning LLDB Commands 21

Chapter 1: Getting Started 22

Chapter 2: Help & Apropos 36

Chapter 3: Attaching with LLDB 40

Chapter 4: Stopping in Code 48

Chapter 5: Expression 65

Chapter 6: Thread, Frame & Stepping Around 79

Chapter 7: Image 89

Chapter 8: Watchpoints 105

Chapter 9: Persisting & Customizing Commands 115

Chapter 10: Regex Commands 120

Section II: Understanding Assembly 128

Chapter 11: Assembly Register Calling Convention 129

Chapter 12: Assembly & Memory 148

Chapter 13: Assembly & the Stack 164

Section III: Low Level 185

Chapter 14: Hello, Ptrace 186

Chapter 15: Dynamic Frameworks 196
Chapter 16: Hooking & Executing Code with dlopen & dlsym 211
Chapter 17: Exploring & Method Swizzling Objective-C Frameworks 229
Chapter 18: Hello, Mach-O 252
Chapter 19: Mach-O Fun 273
Chapter 20: Code Signing 291
Section IV: Custom LLDB Commands 313
Chapter 21: Hello, Script Bridging 315
Chapter 22: Debugging Script Bridging 325
Chapter 23: Script Bridging Classes & Hierarchy 340
Chapter 24: Script Bridging with Options & Arguments 360
Chapter 25: Script Bridging with SBValue & Memory 381
Chapter 26: SB Examples, Improved Lookup ... 404
Chapter 27: SB Examples, Resymbolicating a Stripped ObjC Binary 420
Chapter 28: SB Examples, Malloc Logging 435
Section V: DTrace 459

Chapter 29: Hello, DTrace 460
Chapter 30: Intermediate DTrace 479
Chapter 31: DTrace vs. objc_msgSend 496
Appendix A: LLDB Cheatsheet 523
Appendix B: Python Environment Setup 530
Appendix C: LLDB Bug 534
Conclusion 535
Want to Grow Your Skills? 537

Table of Contents: Extended

Introduction 16
What you need 17
Who this book is for 17
Book source code and forums 18
Book updates 18
Custom LLDB scripts repo 18
License 19
Acknowledgments 20
About the cover 20

Section I: Beginning LLDB Commands 21

Chapter 1: Getting Started 22
Getting around Rootless 22
Attaching LLDB to Xcode 25
Where to go from here? 35

Chapter 2: Help & Apropos 36
The "help" command 36
The "apropos" command 38
Where to go from here? 39

Chapter 3: Attaching with LLDB 40
Where to go from here? 47

Chapter 4: Stopping in Code 48
Signals 48
LLDB breakpoint syntax 52
Finally... creating breakpoints 56
Where to go from here? 64

Chapter 5: Expression 65
Formatting p and po 65

Swift vs Objective-C debugging contexts 70
User defined variables 71
Where to go from here? 78
Chapter 6: Thread, Frame & Stepping Around 79
Stack 101 79
Examining the stack's frames 80
Stepping 83
Examining data in the stack 86
Where to go from here? 88
Chapter 7: Image 89
Wait... modules? 89
Snooping around 98
Where to go from here? 103
Chapter 8: Watchpoints 105
Watchpoint best practices 105
Finding a property's offset 106
What caused the watchpoint 109
The Xcode GUI watchpoint equivalent 111
Where to go from here? 114
Chapter 9: Persisting & Customizing Commands .. 115
Persisting... how? 115
Creating the .lldbinit file 116
Command aliases with arguments 118
Where to go from here? 119
Chapter 10: Regex Commands 120
command regex 120
Executing complex logic 122
Chaining regex inputs 123
Supplying multiple parameters 126
Where to go from here? 127

Section II: Understanding Assembly 128

Chapter 11: Assembly Register Calling Convention 129

Assembly 101 130
x86_64 register calling convention 132
Objective-C and registers 134
Putting theory to practice 135
Swift and registers 140
RAX, the return register 142
Changing around values in registers 144
Where to go from here? 147

Chapter 12: Assembly & Memory 148

Setting up the Intel-Flavored Assembly Experience™ 148
The RIP register 152
Registers and breaking up the bits 155
Breaking down the memory 158
Endianness... this stuff is reversed? 161
Where to go from here? 163

Chapter 13: Assembly & the Stack 164

The stack, revisited 164
Stack pointer & base pointer registers 165
Stack related opcodes 168
Observing RBP & RSP in action 170
The stack and 7+ parameters 177
The stack and debugging info 180
Stack exploration takeaways 182
Where to go from here? 183

Section III: Low Level 185

Chapter 14: Hello, Ptrace 186

System calls 186

The foundation of attachment, ptrace 187
ptrace arguments 189
Creating attachment issues 192
Getting around PT_DENY_ATTACH 192
Other anti-debugging techniques 195
Where to go from here? 195

Chapter 15: Dynamic Frameworks 196
Why dynamic frameworks? 196
Statically inspecting an executable's frameworks 197
Modifying the load commands 201
Loading frameworks at runtime 204
Exploring frameworks 206
Loading frameworks on an actual iOS device 208
Where to go from here? 210

Chapter 16: Hooking & Executing Code with dlopen & dlsym 211
The Objective-C runtime vs. Swift & C 211
Setting up your project 212
Easy mode: hooking C functions 213
Hard mode: hooking Swift methods 222
Where to go from here? 228

Chapter 17: Exploring & Method Swizzling Objective-C Frameworks 229
Between iOS 10 and 12 230
Sidestepping checks in prepareDebuggingOverlay 238
Introducing method swizzling 243
Where to go from here? 250

Chapter 18: Hello, Mach-O 252
Terminology 253
The Mach-O header 255
The load commands 263

Segments 265
Programmatically finding segments and sections 267
Where to go from here? 272

Chapter 19: Mach-O Fun 273
Mach-O Refresher 273
The Mach-O sections 274
Finding HTTP strings 277
Sections in the __DATA segment 281
Cheating freemium games 283
Where to go from here? 290

Chapter 20: Code Signing 291
Setting up 292
Terminology 293
Public/private keys 293
Entitlements 296
Provisioning profiles 298
Exploring the WordPress app 300
Resigning the WordPress app 303
Where to go from here? 312

Section IV: Custom LLDB Commands 313

Chapter 21: Hello, Script Bridging 315
Credit where credit's due 316
Python 101 316
Creating your first LLDB Python script 320
Setting up commands efficiently 322
Where to go from here? 324

Chapter 22: Debugging Script Bridging 325
Debugging your debugging scripts with pdb 326
pdb's post mortem debugging 328
expression's Debug Option 334
How to handle problems 336

Where to go from here? 339

Chapter 23: Script Bridging Classes & Hierarchy... 340

The essential classes 340
Learning & finding documentation on script bridging classes 346
Creating the BreakAfterRegex command 348
Where to go from here? 359

Chapter 24: Script Bridging with Options & Arguments 360

Setting up 361
The optparse Python module 363
Adding options without params 364
Adding options with params 371
Passing parameters into the breakpoint callback function 374
Where to go from here? 380

Chapter 25: Script Bridging with SBValue & Memory 381

A detour down memory layout lane 382
SBValue 394
lldb.value 402
Where to go from here? 403

Chapter 26: SB Examples, Improved Lookup 404

Automating script creation 405
lldbinit directory structure suggestions 406
Implementing the lookup command 408
Adding options to lookup 416
Where to go from here? 419

Chapter 27: SB Examples, Resymbolicating a Stripped ObjC Binary 420

So how are you doing this, exactly? 421
50 Shades of Ray 422

The "stripped" 50 Shades of Ray 429
Building sbt.py 431
Implementing the code 432
Where to go from here? 434

Chapter 28: SB Examples, Malloc Logging 435
Setting up the scripts 435
MallocStackLogging explained 436
Hunting in getenv 440
Testing the functions 445
Turning numbers into stack frames 450
Stack trace from a Swift object 452
DRY Python code 454
Where to go from here? 458

Section V: DTrace 459

Chapter 29: Hello, DTrace 460
The bad news 460
Jumping right in 461
DTrace Terminology 465
Learning while listing probes 468
A script that makes DTrace scripts 470
Where to go from here? 478

Chapter 30: Intermediate DTrace 479
Getting started 479
DTrace & Swift in theory 480
DTrace variables & control flow 483
Inspecting process memory 487
Playing with open syscalls 489
DTrace & destructive actions 491
Where to go from here? 495

Chapter 31: DTrace vs. objc_msgSend 496
Building your proof-of-concept 496

How to get around no probes in a stripped binary 501
Researching method calls using... DTrace! 502
Scary assembly, part II 510
Converting research into code 512
Limiting scope with LLDB 517
Fixing up the snoopie script 520
Where to go from here? 521

Appendix A: LLDB Cheatsheet 523
Getting help 523
Finding code 523
Breakpoints 524
Expressions 525
Stepping 526
GDB formatting 527
Memory 527
Registers & assembly 528
Modules 529

Appendix B: Python Environment Setup 530
Getting Python 530
Python text editors 531
Working with the LLDB Python module 533

Appendix C: LLDB Bug 534
Solution 534

Conclusion 535

Want to Grow Your Skills? 537

Introduction

Debugging has a rather bad reputation. I mean, if the developer had a complete understanding of the program, there wouldn't be any bugs and they wouldn't be debugging in the first place, right?

Don't think like that.

There are always going to be bugs in your software — or any software, for that matter. No amount of test coverage imposed by your product manager is going to fix that. In fact, viewing debugging as *just* a process of fixing something that's broken is actually a poisonous way of thinking that will mentally hinder your analytical abilities.

Instead, you should view debugging as simply **a process to better understand a program**. It's a subtle difference, but if you truly believe it, any previous drudgery of debugging simply disappears.

The same negative connotation can also be applied to reverse engineering software. Images of masked hackers stealing bank accounts and credit cards may come to mind, but for this book, reverse engineering really is just debugging without source code — which in turn helps you gain a better understanding of a program or system.

There's nothing wrong with reverse engineering in itself. In fact if debugging was a game, then reverse engineering is simply debugging on the "difficult" setting — which is quite a fun setting if you've been playing the game for a while. :]

In this book, you'll come to realize debugging is an enjoyable process to help you better understand software. Not only will you learn to find bugs faster, but you'll also learn how other developers have solved problems similar to yours. You'll also learn how to create custom, powerful debugging scripts that will help you quickly find answers to any item that piques your interest, whether it's in your code — or someone else's.

What you need

To follow along with the tutorials in this book, you'll need the following:

- A Mac running **Mojave** (10.14) or later. Earlier versions might work, but they're untested.
- **Xcode 10.0 or later.** Packaged with Xcode is the latest and greatest version of **LLDB**, the debugger you'll use extensively throughout this book. At the time of this writing, the version of LLDB packaged with Xcode is **lldb-1000.11.37.1**.
- **Python 2.7**. LLDB uses Python 2.7 to run its Python scripts. Fortunately, Python 2.7 automatically ships with macOS, as well as with Xcode. You can verify you have the correct version installed by typing `python --version` in `Terminal`.
- **A 64 bit iOS device running iOS 12 or later, and a paid membership to the iOS development program [optional]**. For most chapters in the book, you can run any iOS programs in the Simulator. However, you'll get more out of this book by using a 64-bit iOS device to test out certain ideas or suggestions littered throughout the book.

Once you have these items in place, you'll be able to follow along with almost every chapter in this book. For certain sections, you'll need to disable the **Rootless** security feature in order to use some of the tools (i.e. **DTrace**). This is discussed in Chapter 1.

Who this book is for

The art of debugging code should really be studied by every developer. However, there will be some of you that will get more out of this book. This book is written for:

- Developers who want to become better at debugging with LLDB
- Developers who want to build complex debugging commands with LLDB
- Developers who want to take a deeper dive into internals of Swift and Objective-C
- Developers who are interested in understanding what they can do to their program through reverse engineering
- Developers who are interested in modern, proactive reverse engineering strategies
- Developers who want to be confident in finding answers to questions they have about their computer or software

This book is for intermediate to advanced developers who want to take their debugging and code exploration game to the next level.

Book source code and forums

This book comes with the source code, Python scripts, starter and completed projects for each chapter. These resources are shipped with the PDF.

We've also set up an official forum for the book at forums.raywenderlich.com. This is a great place to ask questions about the book, discuss debugging strategies or to submit any errors you may find.

Book updates

Great news: since you purchased the PDF version of this book, you'll receive free updates of the book's content!

The best way to receive update notifications is to sign up for our weekly newsletter. This includes a list of the tutorials published on raywenderlich.com in the past week, important news items such as book updates or new books, and a few of our favorite developer links. Sign up here:

- www.raywenderlich.com/newsletter

Custom LLDB scripts repo

Finally, you can find a repo of interesting LLDB Python scripts here:

https://github.com/DerekSelander/LLDB

These scripts will help aid in your debugging/reverse engineering sessions and provide novel ideas for your own LLDB scripts.

License

By purchasing *Advanced Apple Debugging & Reverse Engineering*, you have the following license:

- You're allowed to use and/or modify the source code in *Advanced Apple Debugging & Reverse Engineering* in as many applications as you want, with no attribution required.
- You're allowed to use and/or modify all art, images, or designs that are included in *Advanced Apple Debugging & Reverse Engineering* in as many applications as you want, but must include this attribution line somewhere inside your game: "Artwork/images/designs: from the *Advanced Apple Debugging & Reverse Engineering* book, available at www.raywenderlich.com".
- The source code included in *Advanced Apple Debugging & Reverse Engineering* is for your own personal use only. You're **NOT** allowed to distribute or sell the source code in *Advanced Apple Debugging & Reverse Engineering* without prior authorization.
- This book is for your own personal use only. You're **NOT** allowed to sell this book without prior authorization, or distribute it to friends, co-workers, or students; they must to purchase their own copy instead.

All materials provided with this book are provided on an "as is" basis, without warranty of any kind, express or implied, including but not limited to the warranties of merchantability, fitness for a particular purpose and noninfringement. In no event shall the authors or copyright holders be liable for any claim, damages or other liability, whether in an action of contract, tort or otherwise, arising from, out of or in connection with the software or the use or other dealings in the software.

All trademarks and registered trademarks appearing in this guide are the property of their respective owners.

Acknowledgments

We would like to thank many people for their assistance in making this possible:

- **Our families:** For bearing with us in this crazy time as we worked all hours of the night to get this book ready for publication!
- **Everyone at Apple:** For developing an amazing platform, for constantly inspiring us to improve our games and skill sets and for making it possible for many developers to make a living doing what they love!
- **And most importantly, the readers of raywenderlich.com — especially you!** Thank you so much for reading our site and purchasing this book. Your continued readership and support is what makes all of this possible!

About the cover

The Wana (pronounced "vah-na") is a sea urchin native to the Indo-West Pacific region. This sea urchin has two types of spines: a longer hollow spine and a shorter, toxin producing spine. The Wana contains light-sensitive nerves on its skin which can detect potential threats and can move its spines accordingly towards the threat.

Much like finding bugs in a program, stepping on one of these creatures really, really sucks. Pain settles in (inflicted by either your product manager or the sea urchin) and can last up to several hours, even though the issue may remain for an extended period of time. In addition, just like bugs in a program, if you find one of these lovely creatures, there are usually many more in close proximity!

Section I: Beginning LLDB Commands

This section will cover the basics of using LLDB, Apple's software debugger. You'll explore an application named **Signals**, an Objective-C/Swift application that illustrates how Unix signals can be processed within an application. You'll learn some strategies to find and create Swift syntax-style breakpoints as well as Objective-C style breakpoints. By the end of this section, you'll be able to wield the debugger to perform most of the basic tasks needed for debugging, as well as create your own simple custom commands.

Chapter 1: Getting Started

Chapter 2: Help & Apropos

Chapter 3: Attaching with LLDB

Chapter 4: Stopping in Code

Chapter 5: Expression

Chapter 6: Thread, Frame & Stepping Around

Chapter 7: Image

Chapter 8: Watchpoints

Chapter 9: Persisting & Customizing Commands

Chapter 10: Regex Commands

1 Chapter 1: Getting Started

In this chapter, you're going to get acquainted with LLDB and investigate the process of introspecting and debugging a program. You'll start off by introspecting a program you didn't even write — Xcode!

You'll take a whirlwind tour of a debugging session using LLDB and discover the amazing changes you can make to a program you've absolutely zero source code for. This first chapter heavily favors doing over learning, so a lot of the concepts and deep dives into certain LLDB functionality will be saved for later chapters.

Let's get started.

Getting around Rootless

Before you can start working with LLDB, you need to learn about a feature introduced by Apple to thwart malware. Unfortunately, this feature will *also* thwart your attempts to introspect and debug using LLDB and other tools like DTrace. Never fear though, because Apple included a way to turn this feature off — for those who know what they're doing. And you're going to become one of these people who knows what they're doing!

The feature blocking your introspection and debugging attempts is **System Integrity Protection**, also known as **Rootless**. This system restricts what programs can do — even if they have root access — to stop malware from planting itself deep inside your system.

Although Rootless is a substantial leap forward in security, it introduces some annoyances as it makes programs harder to debug. Specifically, it prevents other processes from attaching a debugger to programs Apple signs.

Since this book involves debugging not only your own applications, but any application you're curious about, it's important that you remove this feature while you learn about debugging so you can inspect any application of your choosing.

If you currently have Rootless enabled, you'll be unable to attach to the majority of Apple's programs.

For example, try attaching LLDB to the `Finder` application.

Open up a Terminal window and look for the Finder process, like so:

```
lldb -n Finder
```

You'll notice the following error:

```
error: attach failed: cannot attach to process due to System Integrity
Protection
```

> **Note**: There are many ways to attach to a process, as well as specific configurations when LLDB attaches successfully. To learn more about attaching to a process, check out Chapter 3, "Attaching with LLDB".

Disabling Rootless

> **Note**: A safer way to follow along with this book would be to create a dedicated virtual machine using **VMWare** or **VirtualBox** and disable Rootless on that VM following the steps detailed below. Downloading and setting up a macOS VM can take about an hour depending on your computer's hardware (and internet speed!). Get the latest installation virtual machine instructions from Google since the macOS version and VM software will have different installation steps. If you choose to disable Rootless on your computer without a VM, it would be ideal to re-enable Rootless once you're done with that particular chapter. Fortunately, there's only a handful of chapters in this book that require Rootless to be disabled!

To disable Rootless, perform the following steps:

1. Restart your macOS machine.
2. When the screen turns blank, hold down **Command** + **R** until the Apple boot logo appears. This will put your computer into **Recovery Mode**.
3. Now, find the **Utilities** menu from the top and then select **Terminal**.

4. With the Terminal window open, type:

```
csrutil disable && reboot
```

5. Provided the `csrutil disable` command succeeded, your computer will restart with Rootless disabled.

You can verify if you've successfully disabled Rootless by querying its status in Terminal once your computer starts up by typing:

```
csrutil status
```

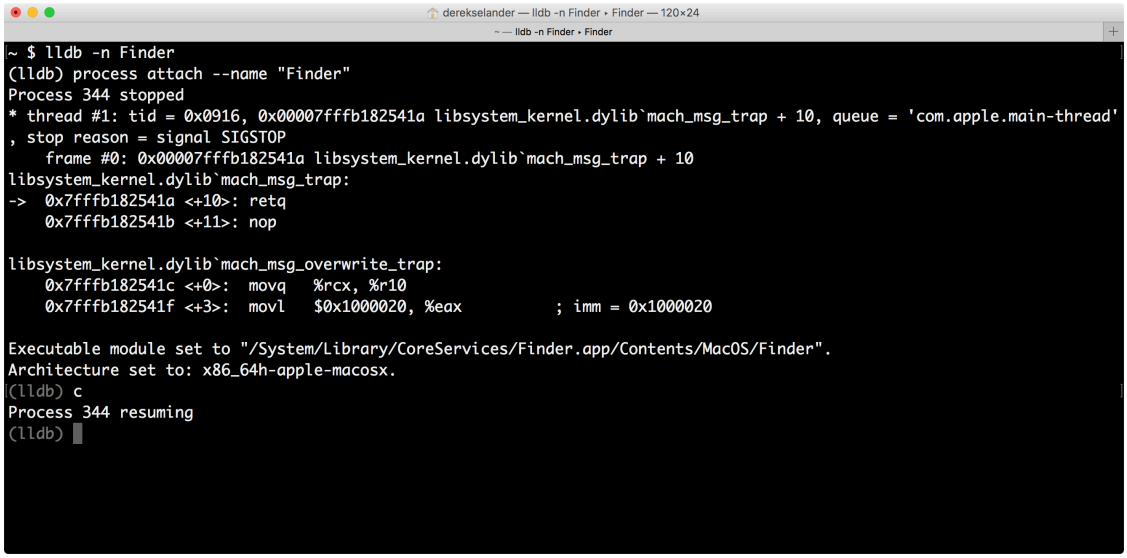
You should see the following:

```
System Integrity Protection status: disabled.
```

Now that SIP is disabled, perform the same "Attach to Finder" LLDB command you tried earlier.

```
lldb -n Finder
```

LLDB should now attach itself to the current Finder process. The output of a successful attach should look like this:

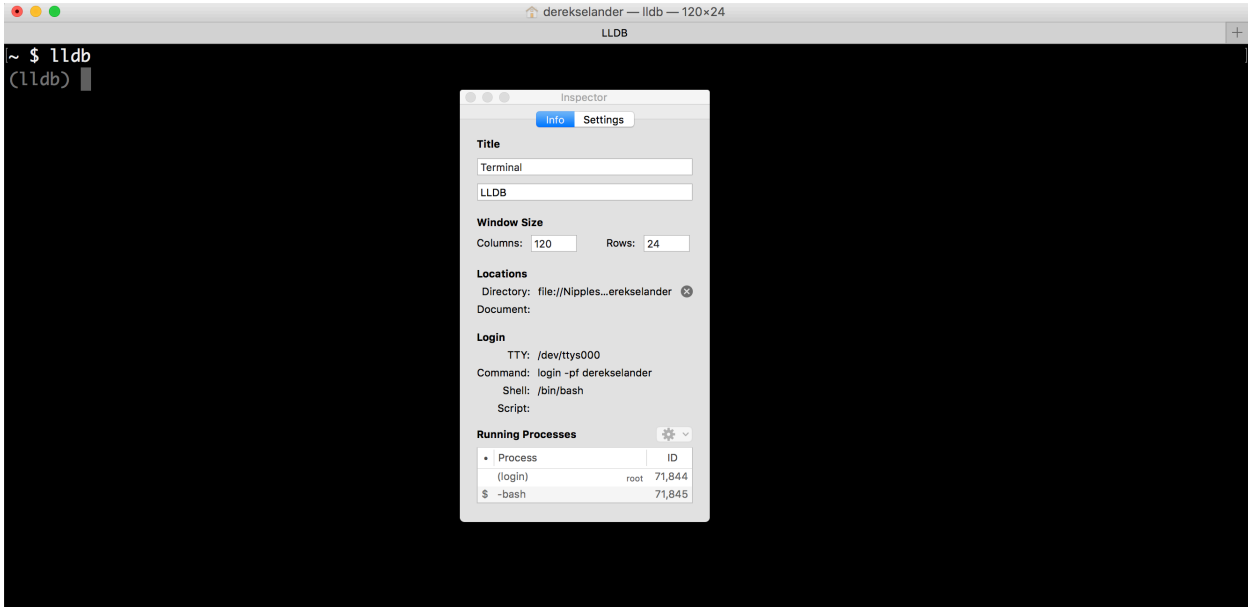


After verifying a successful attach, detach LLDB by either killing the Terminal window, or typing `quit` and confirming in the LLDB console.

Attaching LLDB to Xcode

Now that you've disabled Rootless, you can attach LLDB to any process on your macOS machine (some hurdles may apply, such as with `ptrace` system call, but we'll get to that later). You're first going to look into an application you frequently use in your day-to-day development: Xcode! Make sure you have the latest version of **Xcode 10** installed on your computer before continuing.

Open a new Terminal window. Next, edit the Terminal tab's title by pressing ⌘ + **Shift** + **I**. A new popup window will appear. Edit the **Tab Title** to be **LLDB**.



Next, make sure Xcode isn't running, or you'll end up with multiple running instances of Xcode, which could cause confusion.

In Terminal, type the following:

```
lldb
```

This launches LLDB.

Create a new Terminal tab by pressing ⌘ + **T**. Edit the tab's title again using ⌘ + **Shift** + **I** and name the tab **Xcode stderr**. This Terminal tab will contain all output when you print content from the debugger.

Make sure you are on the **Xcode stderr** Terminal tab and type the following:

```
tty
```

You should see something similar to below:

```
/dev/ttys027
```

Don't worry if yours is different; I'd be surprised if it wasn't. Think of this as the address to your Terminal session.

To illustrate what you'll do with the Xcode stderr tab, create yet another tab and type the following into it:

```
echo "hello debugger" 1>/dev/ttys027
```

Be sure to replace your Terminal path with your unique one obtained from the `tty` command.

Now switch back to the **Xcode stderr** tab. The words `hello debugger` should have popped up. You'll use the same trick to pipe the output of Xcode's stderr to this tab.

Finally, close the third, unnamed tab and navigate back to the LLDB tab.

To summarize: You should now have two Terminal tabs: a tab named "LLDB", which contains an instance of LLDB running, and another tab named "Xcode stderr", which contains the `tty` command you performed earlier.

From there, enter the following into the LLDB Terminal tab:

```
(lldb) file /Applications/Xcode.app/Contents/MacOS/Xcode
```

This will set the executable target to Xcode.

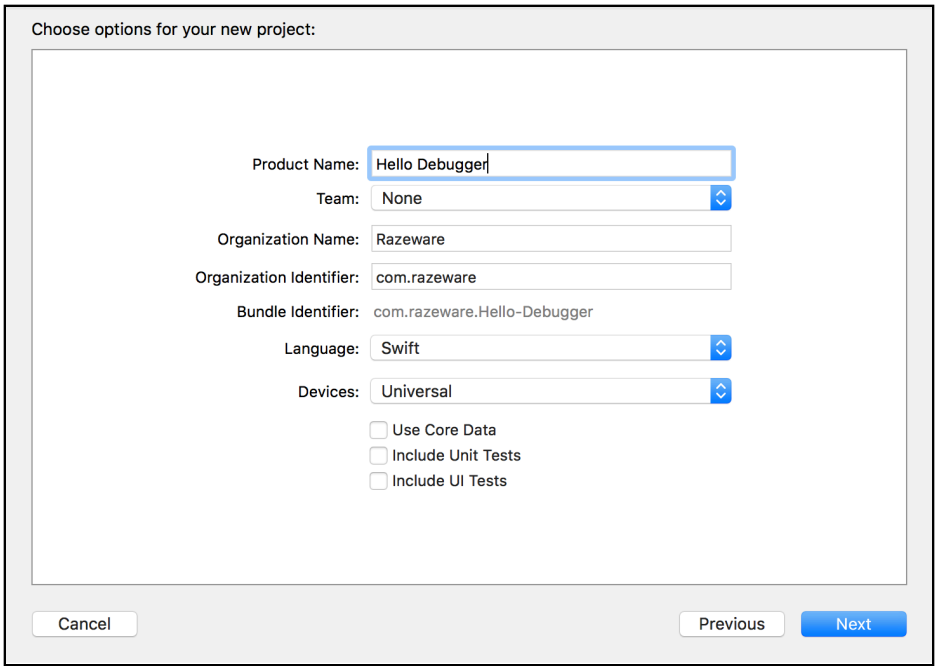
> **Note**: If you are using a prerelease version of Xcode, then the name and path of Xcode could be different.
>
> You can check the path of the Xcode you are currently running by launching Xcode and typing the following in Terminal:
>
> `ps -ef `pgrep -x Xcode``
>
> Once you have the path of Xcode, use that new path instead.

Now launch the Xcode process from LLDB, replacing `/dev/ttys027` with your Xcode stderr tab's `tty` address again:

```
(lldb) process launch -e /dev/ttys027 --
```

The launch argument `e` specifies the location of `stderr`. Common logging functionality, such as Objective-C's `NSLog` or Swift's `print` function, outputs to stderr — yes, not stdout! You'll print your own logging to stderr later.

Xcode will launch after a moment. Switch over to Xcode and click **File ▸ New ▸ Project...**. Next, select **iOS ▸ Application ▸ Single View Application** and click **Next**. Name the product **Hello Debugger**. Make sure to select **Swift** as the programming language and deselect any options for Unit or UI tests. Click **Next** and save the project wherever you wish.



You now have a new Xcode project. Arrange the windows so you can see both Terminal and Xcode.

Navigate to Xcode and open **ViewController.swift**.

> **Note**: You might notice some output on the Xcode stderr Terminal window; this is due to content logged by the authors of Xcode via NSLog or another stderr console printing function.

A "Swiftly" changing landscape

Apple has been cautious in its adoption of Swift in its own software — and understandably so. No matter one's (seemingly religious) beliefs on Swift, for better or worse, it's still an immature language moving at an incredible pace with breaking changes abound.

However, things are a changin' at Apple. Apple is now more aggressively adopting Swift in their own applications such as the iOS Simulator... and even Xcode!

At last count, Xcode 10 includes almost *200* frameworks which contain Swift.

How can you verify this information yourself? This info was obtained using a combination of my helper LLDB scripts found here: https://github.com/DerekSelander/LLDB which are free to all. I would strongly recommend that you clone and install this repo, as I'll occasionally push out new LLDB commands that make debugging much more enjoyable. Installation instructions are in the README of the repo. I'll refer to this repo throughout the book when there's a situation that's significantly easier through these LLDB scripts.

The scary command to obtain this information is the following. You'll need to install the repo mentioned in the note above if you wish to execute this command.

```
(lldb) sys echo "$(dclass -t swift)" | grep -v _ | grep "\." | cut -d. -
f1 | uniq | wc -l
```

Breaking this command down, the **`dclass -t swift`** command is a custom LLDB command that will dump all classes known to the process that are Swift classes. The **`sys`** command will allow you to execute commands like you were in Terminal, but anything in the `$()` will get evaluated first via LLDB. From there, it's a matter of manipulating the output of all the Swift classes given by the `dclass` command.

Swift class naming will typically have the form `ModuleName.ClassName` where the module is the framework that the class is implemented in. The rest of the command does the following:

- `grep -v _`: Exclude any Swift names that include an underscore, which is a typical trait of the class names in the Swift standard library.
- `grep "\."`: Filter by Swift classes that contain a period in the class name.
- `cut -d. -f1`: Isolate the module name before the period.
- `uniq`: Then grab all unique values of the modules.
- `wc -l`: and get the count of it.

These custom LLDB commands (`dclass`, `sys`) were built using Python along with LLDB's Python module (confusingly also called `lldb`). You'll get very accustomed to working with this Python module in section IV of this book as you learn to build custom, advanced LLDB scripts.

Finding a class with a click

Now that Xcode is set up and your Terminal debugging windows are correctly created and positioned, it's time to start exploring Xcode using the help of the debugger.

While debugging, knowledge of the Cocoa SDK can be extremely helpful. For example, **`-[NSView hitTest:]`** is a useful Objective-C method that returns the class responsible for the handled click or gesture for an event in the run loop. This method will first be triggered on the containing `NSView` and recursively drill into the furthest subview that handles this touch. You can use this knowledge of the Cocoa SDK to help determine the class of the view you've clicked on.

In your LLDB tab, type `Ctrl + C` to pause the debugger. From there, type:

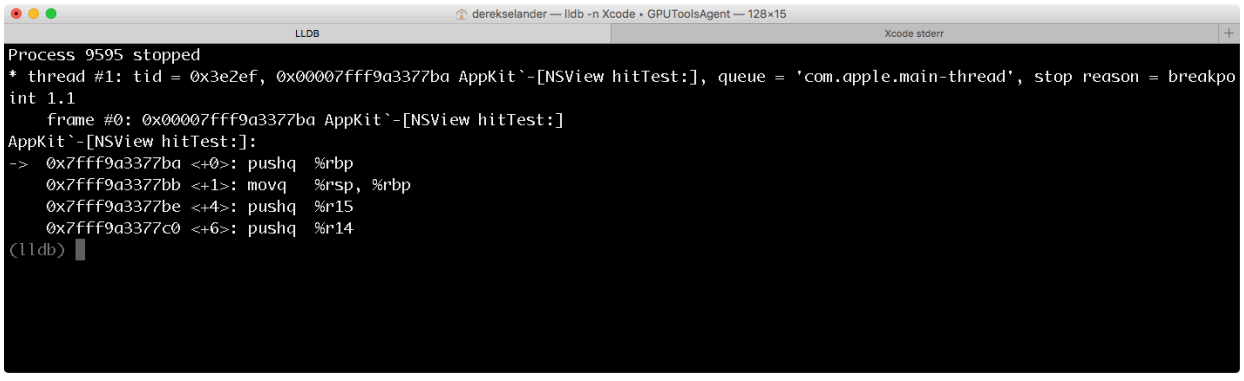
```
(lldb) b -[NSView hitTest:]
Breakpoint 1: where = AppKit`-[NSView hitTest:], address =
0x000000010338277b
```

This is your first breakpoint of many to come. You'll learn the details of how to create, modify, and delete breakpoints in Chapter 4, "Stopping in Code", but for now simply know you've created a breakpoint on `-[NSView hitTest:]`.

Xcode is now paused thanks to the debugger. Resume the program:

```
(lldb) continue
```

Click anywhere in the Xcode window (or in some cases even moving your cursor over Xcode will do the same); Xcode will instantly pause and LLDB will indicate a breakpoint has been hit.



The `hitTest:` breakpoint has fired. You can inspect which view was hit by inspecting the **RDI** CPU register. Print it out in LLDB:

```
(lldb) po $rdi
```

This command instructs LLDB to print out the contents of the object at the memory address referenced by what's stored in the `RDI` assembly register.

> **Note**: Wondering why the command is `po`? `po` stands for *print object*. There's also `p`, which simply prints the contents of `RDI`. `po` is usually more useful as it gives the `NSObject`'s (or Swift's `SwiftObject`'s) `description` or `debugDescription` methods, if available.

Assembly is an important skill to learn if you want to take your debugging to the next level. It will give you insight into Apple's code — even when you don't have any source code to read from. It will give you a greater appreciation of how the Swift compiler team danced in and out of Objective-C with Swift, and it will give you a greater appreciation of how everything works on your Apple devices.

You'll learn more about registers and assembly in Chapter 11, "Assembly Register Calling Convention".

For now, simply know the `$rdi` register in the above LLDB command contains the instance of the subclass `NSView` the `hitTest:` method was called upon.

> **Note**: The output will produce different results depending on where you clicked and what version of Xcode you're using. It could give a private class specific to Xcode, or it could give you a public class belonging to Cocoa.

In LLDB, type the following to resume the program:

```
(lldb) continue
```

Instead of continuing, Xcode will likely hit another breakpoint for `hitTest:` and pause execution. This is due to the fact that the `hitTest:` method is recursively calling this method for all subviews contained within the parent view that was clicked. You can inspect the contents of this breakpoint, but this will soon become tedious since there are so many views that make up Xcode.

Automate the hitTest:

The process of clicking on a view, stopping, `po`'ing the RDI register then continuing can get tiring quickly. What if you created a breakpoint to automate all of this?

There's several ways to achieve this, but perhaps the cleanest way is to declare a new breakpoint with all the traits you want. Wouldn't that be neat?! :]

Remove the previous breakpoint with the following command:

```
(lldb) breakpoint delete
```

LLDB will ask if you sure you want to delete all breakpoints, either press enter or press 'Y' then enter to confirm.

Now, create a new breakpoint with the following:

```
(lldb) breakpoint set -n  "-[NSView hitTest:]" -C "po $rdi" -G1
```

The gist of this command says to create a breakpoint on `-[NSView hitTest:]`, have it execute the "`po $rdi`" command, then automatically continue after executing the command. You'll learn more about these options in a later chapter.

Resume execution with the `continue` command:

```
(lldb) continue
```

Now, click anywhere in Xcode and check out the output in the Terminal console. You'll see many many `NSViews` being called to see if they should take the mouse click!

Filter breakpoints for important content

Since there are so many `NSViews` that make up Xcode, you need a way to filter out some of the noise and only stop on the `NSView` relevant to what you're looking for. This is an example of debugging a frequently-called method, where you want to find a unique case that helps pinpoint what you're really looking for.

As of Xcode 10, the class responsible for visually displaying your code in the Xcode IDE is a private Swift class belonging to the `IDESourceEditor` module, named **`IDESourceEditorView`**. This class acts as the visual coordinator to hand off all your code to other private classes to help compile and create your applications.

Let's say you want to break only when you click an instance of `IDESourceEditorView`. You can modify the existing breakpoint to stop only on a `IDESourceEditorView` click by using **breakpoint conditions**.

Provided you still have your `-[NSView hitTest:]` breakpoint set, and it's the only active breakpoint in your LLDB session, you can modify that breakpoint with the following LLDB command:

```
(lldb) breakpoint modify -c '(BOOL)[NSStringFromClass((id)[$rdi class])
containsString:@"IDESourceEditorView"]' -G0
```

This command modifies all existing breakpoints in your debugging session and creates a condition which gets evaluated everytime `-[NSView hitTest:]` fires. If the condition evaluates to true, then execution will pause in the debugger. This condition checks that the instance of the `NSView` is of type `IDESourceEditorView`. The final `-G0` says to modify the breakpoint to not automatically resume execution after the action has been performed.

After modifying your breakpoint above, click on the **code area** in Xcode. LLDB should stop on `hitTest:`. Print out the instance of the class this method was called on:

```
(lldb) po $rdi
```

Your output should look something similar to the following:

```
IDESourceEditorView: Frame: (0.0, 0.0, 1109.0, 462.0), Bounds: (0.0, 0.0,
1109.0, 462.0) contentViewOffset: 0.0
```

This is printing out the object's **description**. You'll notice that there is no pointer reference within this, because Swift hides the pointer reference. There's several ways to get around this if you need the pointer reference. The easiest is to use **print formatting**. Type the following in LLDB:

```
(lldb) p/x $rdi
```

You'll get something similar to the following:

```
(unsigned long) $3 = 0x0000000110a42600
```

Since RDI points to a valid Objective-C `NSObject` subclass (written in Swift), you can also get the same info just by `po`'ing this address instead of the register.

Type the following into LLDB while making sure to replace the address with your own:

```
(lldb) po 0x0000000110a42600
```

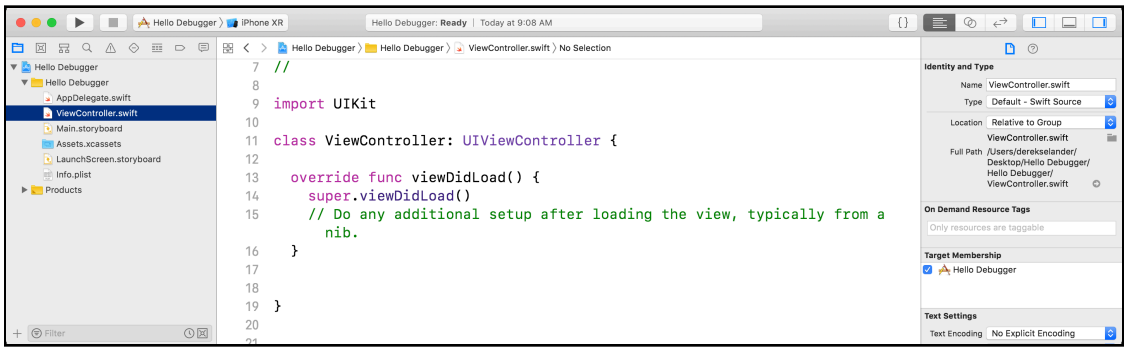
You'll get the same output as earlier.

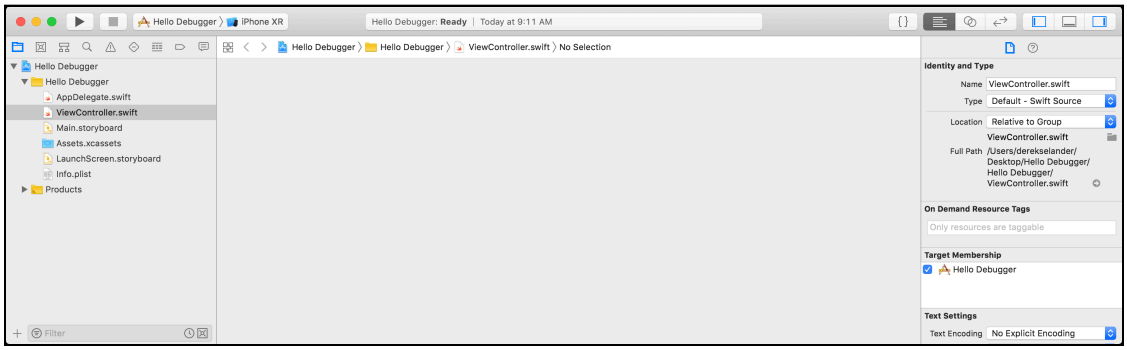
You might be skeptical that this reference pointed at by the RDI register is actually pointing to the `NSView` that displays your code. You can easily verify if that's true or not by typing the following in LLDB:

```
(lldb) po [$rdi setHidden:!(BOOL)[$rdi isHidden]]; [CATransaction flush]
```

> **Note**: Kind of a long command to type out, right? In Chapter 10: "Regex Commands", you'll learn how to build convenient shortcuts so you don't have to type out these long LLDB commands. If you chose to install the LLDB repo mentioned earlier, a convenience command for this above action the **tv** command, or "toggle view"

Provided RDI is pointing to the correct reference, your code editor view will disappear!





You can toggle this view on and off simply by repeatedly pressing `Enter`. LLDB will automatically execute the previous command.

Since this is a subclass of `NSView`, all the methods of `NSView` apply. For example, the `string` command can query the contents of your source code through LLDB. Type the following:

```
(lldb) po [$rdi string]
```

This will dump out the contents of your source code editor. Neat!

Always remember, any APIs that you have in your development cycle can be used in LLDB. If you were crazy enough, you could create an entire app just by executing LLDB comands!

When you get bored of playing with the `NSView` APIs on this instance, copy the address down that RDI is referencing (copy it to your clipboard or add it to the stickies app). You'll reference it again in a second.

Alternatively, did you notice that output preceding the hex value in the `p/x $rdi` command? In my output, I got **$3**, which means that you can use `$3` as a reference for that pointer value you just grabbed. This is incredibly useful when the RDI register points to something else and you still want to reference this `NSView` at a later time.

Swift vs Objective-C debugging

Wait — we're using Objective-C on a Swift class?! You bet! You'll discover that a Swift class is *mostly* all Objective-C underneath the covers (however the same can't be said about Swift structs). You'll confirm this by modifying the console's source code through LLDB using Swift!

First, import the following modules in the Swift debugging context:

```
(lldb) ex -l swift -- import Foundation
(lldb) ex -l swift -- import AppKit
```

The `ex` command (short for `expression`) lets you evaluate code and is the foundation for your `p`/`po` LLDB commands. `-l swift` tells LLDB to interpret your commands as Swift code. You just imported the headers to call appropriate methods in both of these modules through Swift. You'll need these in the next two commands.

Enter the following, replacing `0x0110a42600` with the memory address of your `NSView` subclass you recently copied to your clipboard:

```
(lldb) ex -l swift -o -- unsafeBitCast(0x0110a42600, to: NSView.self)
```

This command prints out the `IDESourceEditorView` instance — but this time using Swift!

Now, add some text to your source code via LLDB:

```
(lldb) ex -l swift -o -- unsafeBitCast(0x0110a42600, to:
NSView.self).insertText("Yay! Swift!")
```

Depending where your cursor was in the Xcode console, you'll see the new string "Yay! Swift!" added to your source code.

When stopping the debugger out of the blue, or on Objective-C code, LLDB will default to using the Objective-C context when debugging. This means the `po` you execute will expect Objective-C syntax unless you force LLDB to use a different language like you

did above. It's possible to alter this, but this book prefers to use Objective-C since the Swift REPL can be brutal for error-checking, has slow compilation times for executing commands, is generally much more buggy, and prevents you from executing methods the Swift LLDB context doesn't know about.

All of this will eventually go away, but we must be patient. The Swift ABI must first stabilize. Only then, can the Swift tooling really become rock solid.

Where to go from here?

This was a breadth-first, whirlwind introduction to using LLDB and attaching to a process where you don't have any source code to aid you. This chapter glossed over a lot of detail, but the goal was to get you right into the debugging/reverse engineering process.

To some, this first chapter might have come off as a little scary, but we'll slow down and describe methods in detail from here on out. There are lots of chapters remaining to get you into the details!

Keep reading to learn the essentials in the remainder of Section 1. Happy debugging!

Chapter 2: Help & Apropos

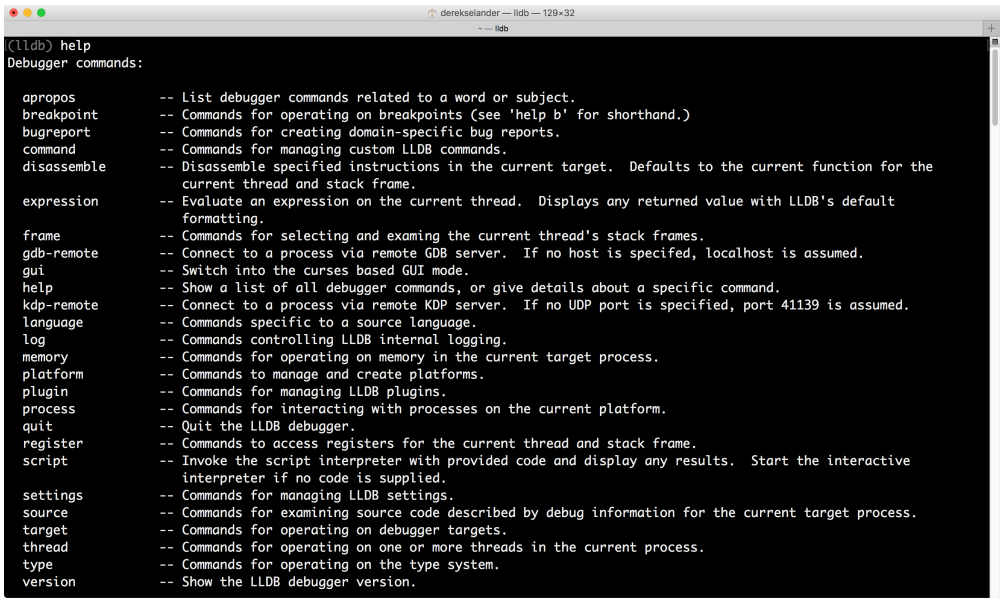
Just like any respectable developer tool, LLDB ships with a healthy amount of documentation. Knowing how to navigate through this documentation — including some of the more obscure command flags — is essential to mastering LLDB.

The "help" command

Open a `Terminal` window and type `lldb`. The LLDB prompt will appear. From there, simply type the `help` command:

```
(lldb) help
```

This will dump out all available commands, including the custom commands loaded from your `~/.lldbinit` — but more on that later.



There's quite a few commands one can use with LLDB.

However, many commands have numerous subcommands, which in turn can have subcommands, which also have their own associated documentation. I told you it was a healthy amount of documentation!

Take the `breakpoint` command for instance. Run the documentation for `breakpoint` by typing the following:

```
(lldb) help breakpoint
```

You'll see the following output:

```
 Commands for operating on breakpoints (see 'help b' for shorthand.)

Syntax: breakpoint

The following subcommands are supported:

      clear   -- Delete or disable breakpoints matching the specified
source file and line.
      command -- Commands for adding, removing and listing LLDB commands
executed when a breakpoint is hit.
      delete  -- Delete the specified breakpoint(s).  If no breakpoints
are specified, delete them all.
      disable -- Disable the specified breakpoint(s) without deleting
them.  If none are specified, disable all breakpoints.
      enable  -- Enable the specified disabled breakpoint(s). If no
breakpoints are specified, enable all of them.
      list    -- List some or all breakpoints at configurable levels of
detail.
      modify  -- Modify the options on a breakpoint or set of breakpoints
in the executable.  If no breakpoint is specified,
                  acts on the last created breakpoint.  With the exception
of -e, -d and -i, passing an empty argument clears
                  the modification.
      name    -- Commands to manage name tags for breakpoints
      read    -- Read and set the breakpoints previously saved to a file
with "breakpoint write".
      set     -- Sets a breakpoint or set of breakpoints in the
executable.
      write   -- Write the breakpoints listed to a file that can be read
in with "breakpoint read".  If given no arguments,
                  writes all breakpoints.

For more help on any particular subcommand, type 'help <command>
<subcommand>'.
```

From there, you can see several supported subcommands. Look up the documentation for `breakpoint name` by typing the following:

```
(lldb) help breakpoint name
```

You'll see the following output:

```
   Commands to manage name tags for breakpoints

Syntax: breakpoint name

The following subcommands are supported:

      add       -- Add a name to the breakpoints provided.
      configure -- Configure the options for the breakpoint name
provided.  If you provide a breakpoint ID, the options will be
                   copied from the breakpoint, otherwise only the options
specified will be set on the name.
      delete    -- Delete a name from the breakpoints provided.
      list      -- List either the names for a breakpoint or info about a
given name.  With no arguments, lists all names

For more help on any particular subcommand, type 'help <command>
<subcommand>'.
```

If you don't understand `breakpoint name` at the moment, don't worry — you'll become familiar with breakpoints and all of the subsequent commands soon. For now, the `help` command is the most important command you can remember.

The "apropos" command

Sometimes you don't know the name of the command you're searching for, but you know a certain word or phrase that might point you in the right direction. The `apropos` command can do this for you; it's a bit like using a search engine to find something on the web.

`apropos` will do a case-insensitive search for any word or string against the LLDB documentation and return any matching results. For example, try searching for anything pertaining to Swift:

```
(lldb) apropos swift
```

You'll see the following output:

```
The following commands may relate to 'swift':
  swift    -- A set of commands for operating on the Swift Language
Runtime.
  demangle -- Demangle a Swift mangled name
  refcount -- Inspect the reference count data for a Swift object

The following settings variables may relate to 'swift':


  target.swift-framework-search-paths -- List of directories to be
```

```
searched when locating frameworks for Swift.
  target.swift-module-search-paths -- List of directories to be searched
when locating modules for Swift.
  target.use-all-compiler-flags -- Try to use compiler flags for all
modules when setting up the Swift expression parser, not just the main
executable.
  target.experimental.swift-create-module-contexts-in-parallel -- Create
the per-module Swift AST contexts in parallel.
```

This dumped everything that might pertain to the word Swift: first the commands, and then the LLDB settings which can be used to control how LLDB operates.

You can also use `apropos` to search for a particular sentence. For example, if you were searching for something that can help with reference counting, you might try the following:

```
(lldb) apropos "reference count"
The following commands may relate to 'reference count':
  refcount -- Inspect the reference count data for a Swift object
```

Notice the quotes surrounding the words `"reference count"`. `apropos` will only accept one argument to search for, so the quotes are necessary to treat the input as a single argument.

Isn't that neat? `apropos` is a handy tool for querying. It's not quite as sophisticated as modern internet search engines; however, with some playing around, you can usually find what you're looking for.

> **Note**: A new bug that popped up in the latest Xcode 10 version of LLDB (lldb-1000.11.37.1) will not give the full command tree when using `apropos`. For example, the `refcount` command displayed above can only actually be used via `language swift refcount`. The `apropos` command displayed this correctly in previous version, (and will likely again in the future) but for now you'll need to do a little digging to get the exact command using `apropos`

Where to go from here?

It's easy to forget the onslaught of LLDB commands that will soon come, but try to commit these two commands, `help` and `apropos`, to heart. They're the foundation for querying information on commands and you'll be using them all the time as you master debugging.

Chapter 3: Attaching with LLDB

Now that you've learned about the two most essential commands, `help` and `apropos`, it's time to investigate how LLDB attaches itself to processes. You'll learn all the different ways you can attach LLDB to processes using various options, as well as what happens behind the scenes when attaching to processes.

The phrase of LLDB "attaching" is actually a bit misleading. A program named `debugserver` (found in `Xcode.app/Contents/SharedFrameworks/LLDB.framework/Resources/`) is responsible for attaching to a target process.

If it's a remote process, such as an iOS, watchOS or tvOS application running on a remote device, a remote `debugserver` gets launched on that remote device. It's LLDB's job to launch, connect, and coordinate with the `debugserver` to handle all the interactions in debugging an application.

Attaching to an existing process

As you've already seen in Chapter 1, you can attach to a process like so:

```
lldb -n Xcode
```

However, there's other ways to do the same thing. You can attach to Xcode by providing the process identifier, or **PID**, of a running program.

> **Note**: Just a reminder, if you didn't disable SIP on your macOS computer, you will not be able to attach LLDB to Apple applications. Attaching to 3rd party applications (even from the App Store!) is still possible as of version 10.14.1 in Mojave provided there are no anti-debugging techniques in the application.

Open Xcode, then open a new `Terminal` session, and finally run the following:

```
pgrep -x Xcode
```

This will output the PID of the Xcode process.

Next, run the following, replacing `89944` with the number output from the command above:

```
lldb -p 89944
```

This tells LLDB to attach to the process with the given PID. In this case, this is your running Xcode process.

Attaching to a future process

The previous command only addresses a running process. If Xcode isn't running, or is already attached to a debugger, the previous commands will fail. How can you catch a process that's about to be launched, if you don't know the PID yet?

You can do that with the `-w` argument, which causes LLDB to wait until a process launches with a PID or executable name matching the criteria supplied using the `-p` or `-n` argument.

For example, kill your existing LLDB session by pressing **Ctrl + D** in your Terminal window then type the following:

```
lldb -n Finder -w
```

This will tell LLDB to attach to the process named Finder whenever it next launches. Next, open a new Terminal tab, and enter the following:

```
pkill Finder
```

This will kill the Finder process and force it to restart. macOS will automatically relaunch Finder when it's killed. Switch back to your first Terminal tab and you'll notice LLDB has now attached itself to the newly created Finder process.

Another way to attach to a process is to specify the path to the executable and manually launch the process at your convenience:

```
lldb -f /System/Library/CoreServices/Finder.app/Contents/MacOS/Finder
```

This will set Finder as the executable to launch. Once you're ready to begin the debug session, simply type the following into the LLDB session:

```
(lldb) process launch
```

> **Note**: An interesting side effect is that `stderr` output (i.e. Swift's `print`, Objective-C's `NSLog`, C's `printf` and company) are automatically sent to the Terminal window when manually launching a process. Other LLDB attaching configurations don't do this automatically.

Options while launching

The `process launch` command comes with a suite of options worth further exploration. If you're curious and want to see the full list of available options for `process launch`, simply type `help process launch`.

Close previous LLDB sessions, open a new Terminal window and type the following:

```
lldb -f /bin/ls
```

This tells LLDB to use `/bin/ls` (the file listing command) as the target executable.

> **Note**: If you omit the `-f` option, LLDB will automatically infer the first argument to be the executable to launch and debug. When debugging Terminal executables, I'll oftentimes type `lldb $(which ls)` (or equivalent), which then gets translated to `lldb /bin/ls`.

You'll see the following output:

```
(lldb) target create "/bin/ls"
Current executable set to '/bin/ls' (x86_64).
```

Since `ls` is a quick program (it launches, does its job, then exits) you'll run this program multiple times with different arguments to explore what each does.

To launch `ls` from LLDB with no arguments. Enter the following:

```
(lldb) process launch
```

You'll see the following output:

```
Process 7681 launched: '/bin/ls' (x86_64)
... # Omitted directory listing output
Process 7681 exited with status = 0 (0x00000000)
```

An `ls` process will launch in the directory you started in. You can change the current working directory by telling LLDB where to launch with the `–w` option. Enter the following:

```
(lldb) process launch –w /Applications
```

This will launch `ls` from within the `/Applications` directory. This is equivalent to the following:

```
$ cd /Applications
$ ls
```

There's yet *another* way to do this. Instead of telling LLDB to change to a directory then run the program, you can pass arguments to the program directly.

Try the following:

```
(lldb) process launch -- /Applications
```

This has the same effect as the previous command, but this time it's doing the following:

```
$ ls /Applications
```

Again, this spits out all your macOS programs, but you specified an argument instead of changing the starting directory. What about specifying your desktop directory as a launch argument? Try running this:

```
(lldb) process launch -- ~/Desktop
```

You'll see the following:

```
Process 8103 launched: '/bin/ls' (x86_64)
ls: ~/Desktop: No such file or directory
Process 8103 exited with status = 1 (0x00000001)
```

Uh-oh, that didn't work. You need the shell to expand the *tilde* in the argument. Try this instead:

```
(lldb) process launch –X true -- ~/Desktop
```

The –X option expands any shell arguments you provide, such as the tilde. There's a shortcut in LLDB for this: simply type `run`. To learn more about creating your own command shortcuts, check out Chapter 9, "Persisting and Customizing Commands".

Type the following to see the documentation for `run`:

```
(lldb) help run
```

You'll see the following:

```
...
Command Options Usage:
  run [<run-args>]


'run' is an abbreviation for 'process launch -X true --'
```

See? It's an abbreviation of the command you just ran! Give the command a go by typing the following:

```
(lldb) run ~/Desktop
```

Environment variables

For Terminal programs, **environment variables** can be equally as important as the program's arguments. If you were to consult the `man 1 ls`, you'll see at that the `ls` command can display output in color so long as the color environment variable is enabled (`CSICOLOR`) and you have the "color pallete" environment variable `LSCOLORS` to tell how to display certain filetypes.

With a target in LLDB, you can launch and set a program with any combination of environement variables.

For example, to display all the environment variables that the `ls` command will launch with, run the following command in LLDB:

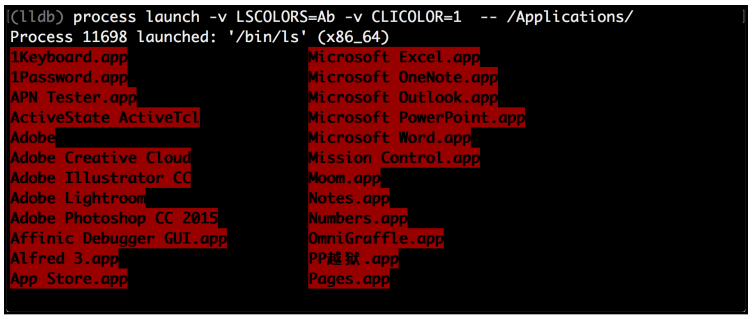
```
(lldb) env
```

This will display all the environment variables for the target. It's important to note that LLDB will not display the environment variables until the target is run at least once. If you don't see any output, just give lldb a simple `run` before executing the `env` command.

You can inspect and augment these environment variables before launch using the `settings set|show|replace|clear|list target.env-vars` command, but you can also just specify them at launch with the `-v` option from the `process launch` command!

Time to display the `/Applications` directory in a horrible red color!

```
(lldb) process launch -v LSCOLORS=Ab -v CLICOLOR=1  -- /Applications/
```



Wow! Doesn't that just burn the eyes? Try a different color with the following:

```
(lldb) process launch -v LSCOLORS=Af -v CLICOLOR=1  -- /Applications/
```

This would equivalent to you executing the following in Terminal without LLDB:

```
LSCOLORS=Af CLICOLOR=1 ls /Applications/
```

Lots of Terminal commands will contain environment variables and their descriptions in the command's `man` page. Always make sure to read about how you'd expect an environment variable to augment a program.

In addition, many commands (and Apple frameworks!) have "private" environment variables not discussed in any documentation or `man` page. You'll look at how you can extract this information out of executables later on in this book.

stdin, stderr, and stout

What about changing the standard streams to a different location? You've already tried changing `stderr` to a different Terminal tab in Chapter 1 using the `-e` flag, but how about `stdout`?

Type the following:

```
(lldb) process launch -o /tmp/ls_output.txt -- /Applications
```

The `-o` option tells LLDB to pipe `stdout` to the given file.

You'll see the following output:

```
Process 15194 launched: '/bin/ls' (x86_64)
Process 15194 exited with status = 0 (0x00000000)
```

Notice there's no output directly from `ls`.

Open another Terminal tab and run the following:

```
cat /tmp/ls_output.txt
```

It's your applications directory output again, as expected!

There is also an option `-i` for `stdin` as well. First, type the following:

```
(lldb) target delete
```

This removes `ls` as the target. Next, type this:

```
(lldb) target create /usr/bin/wc
```

This sets `/usr/bin/wc` as the new target. `wc` can be used to count characters, words or lines in the input given to `stdin`.

You've swapped target executables for your LLDB session from `ls` to `wc`. Now you need some data to provide to `wc`. Open a new Terminal tab and enter the following:

```
echo "hello world" > /tmp/wc_input.txt
```

You'll use this file to give `wc` some input.

Switch back to the LLDB session and enter the following:

```
(lldb) process launch -i /tmp/wc_input.txt
```

You'll see the following output:

```
Process 24511 launched: '/usr/bin/wc' (x86_64)
       1       2      12
Process 24511 exited with status = 0 (0x00000000)
```

This would be functionally equivalent to the following:

```
$ wc < /tmp/wc_input.txt
```

Sometimes you don't want a `stdin` (standard input). This is useful for GUI programs such as Xcode, but doesn't really help for Terminal commands such as `ls` and `wc`.

To illustrate, run the `wc` target with no arguments, like so:

```
(lldb) run
```

The program will just sit there and hang because it's expecting to read something from `stdin`.

Give it some input by typing in `hello world`, press Return, then press **Control + D**, which is the end of transmission character. `wc` will parse the input and exit. You'll see the same output as you did earlier when using the file as the input.

Now, launch the process like this:

```
(lldb) process launch -n
```

You'll see that `wc` exits immediately with the following output:

```
Process 28849 launched: '/usr/bin/wc' (x86_64)
Process 28849 exited with status = 0 (0x00000000)
```

The `-n` option tells LLDB not to create a `stdin`; therefore `wc` has no data to work with and exits immediately.

Where to go from here?

There are a few more interesting options to play with (which you can find via the `help` command), but that's for you to explore on your own time.

For now, try attaching to GUI and non-GUI programs alike. It might seem like you can't understand much without the source code, but you'll find out in the upcoming sections how much information and control you have over these programs.

Chapter 4: Stopping in Code

Whether you're using Swift, Objective-C, C++, C, or an entirely different language in your technology stack, you'll need to learn how to create breakpoints. It's easy to click on the side panel in Xcode to create a breakpoint using the GUI, but the LLDB console can give you much more control over breakpoints.

In this chapter, you're going to learn all about breakpoints and how to create them using LLDB.

Signals

For this chapter, you'll be looking at a project I've supplied; it's called **Signals** and you'll find it in the resources bundle for this chapter.



Open up the **Signals** project using Xcode. **Signals** is a basic master-detail project themed as an American football app that displays some rather nerdily-named offensive play calls.

Internally, this project montors several Unix signals and displays them when the **Signals** program receives them.

Unix signals are a basic form of interprocess communication. For example, one of the signals, `SIGSTOP`, can be used to save the state and pause execution of a process, while its counterpart, `SIGCONT`, is sent to a program to resume execution. Both of these signals can be used by a debugger to pause and continue a program's execution.

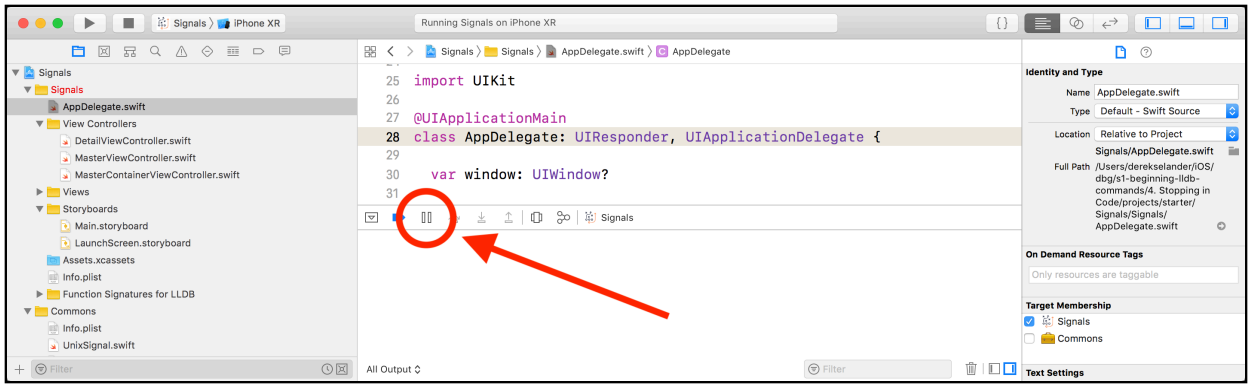
This is an interesting application on several fronts, because it not only explores Unix signal handling, but also highlights what happens when a controlling process (LLDB) handles the passing of Unix signals to the controlled process. By default, LLDB has custom actions for handling different signals. Some signals are not passed onto the controlled process while LLDB is attached.

In order to display a signal, you can either `raise` a Signal from within the application, or send a signal externally from a different application, like Terminal.

In addition, there's a `UISwitch` that toggles the signal handling. When the switch is toggled, it calls a C function `sigprocmask` to disable or enable the signal handlers.

Finally, the Signal application has a **Timeout** bar button which raises the `SIGSTOP` signal from within the application, essentially "freezing" the program. However, if LLDB is attached to the Signals program (and by default it will be, when you build and run through Xcode), calling `SIGSTOP` will allow you to inspect the execution state with LLDB while in Xcode.

Select your iOS Simulator of choice while making sure the iOS Simulator is at least version iOS 12.0 or greater. Build and run the app. Once the project is running, navigate to the Xcode console and pause the debugger.



Resume Xcode and keep an eye on the Simulator. A new row will be added to the UITableView whenever the debugger stops then resumes execution. This is achieved by Signals monitoring the `SIGSTOP` Unix signal event and adding a row to the data model whenever it occurs. When a process is stopped, any new signals will not be immediately processed because the program is sort of, well, stopped.

Xcode breakpoints

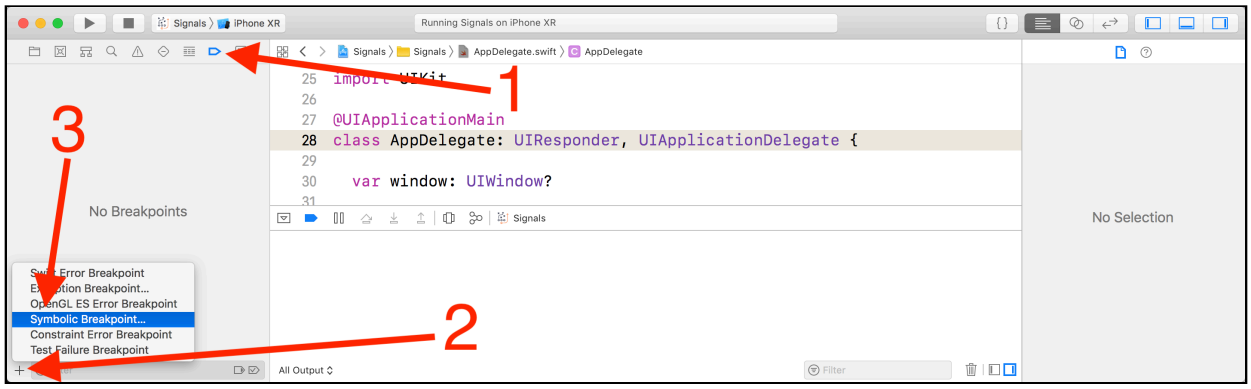
Before you go off learning the cool, shiny breakpoints through the LLDB console, it's worth covering what you can achieve through Xcode alone.

Symbolic breakpoints are a great debugging feature of Xcode. They let you set a breakpoint on a certain **symbol** within your application. An example of a symbol is `-[NSObject init]`, which refers to the `init` method of `NSObject` instances.

The neat thing about symbolic breakpoints in Xcode is that once you enter a symbolic breakpoint, you don't have to type it in again the next time the program launches.

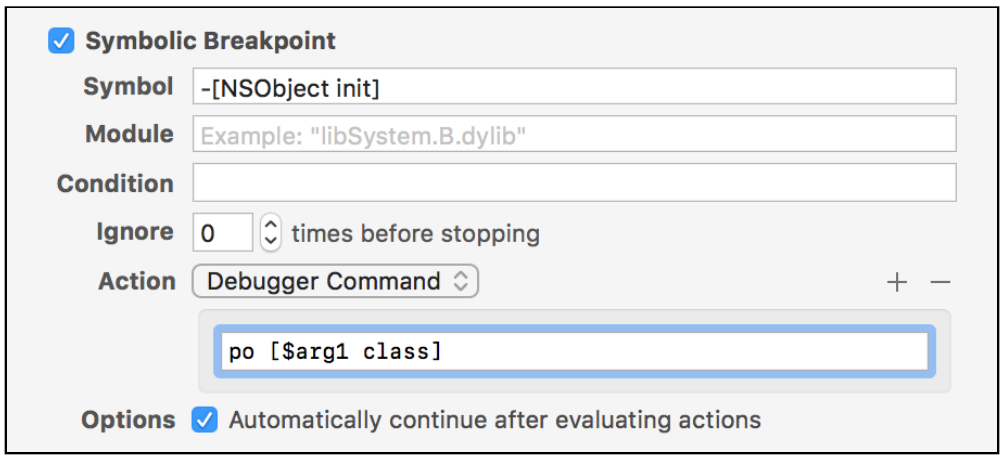
You're now going to try using a symbolic breakpoint to show all the instances of `NSObject` being created.

Kill the app if it's currently running. Next, switch to the **Breakpoint Navigator**. In the bottom left, click the plus button to select the **Symbolic Breakpoint...** option.

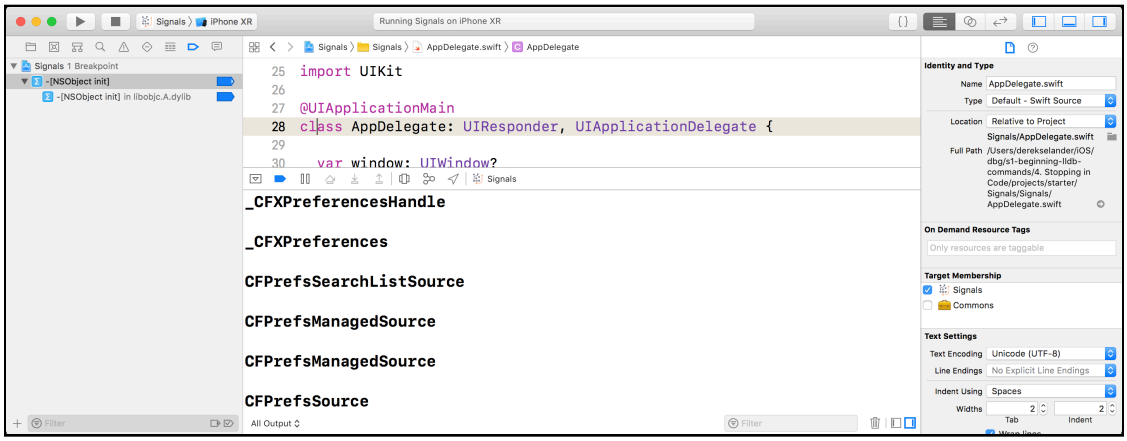


A pop-up will appear. In the **Symbol** part of the popup type: `-[NSObject init]`. Under **Action**, select **Add Action** and then select **Debugger Command** from the dropdown. Next, enter `po [$arg1 class]` in the box below.

Finally, select **Automatically continue after evaluating actions**. Your popup should look similar to below:



Build and run the app. Xcode will dump all the names of the classes it initializes while running the **Signals** program through the console… which, upon viewing, is quite a lot.



What you've done here is set a breakpoint that fires each time `-[NSObject init]` is called. When the breakpoint fires, a command runs in LLDB, and execution of the program continues automatically.

> **Note**: You'll learn how to properly use and manipulate registers in Chapter 11, "Assembly, Registers and Calling Convention", but for now, simply know **`$arg1`** is synonymous to the **`$rdi`** register and can be loosely thought of as holding the instance of a class when `init` is called.

Once you've finished inspecting all the class names dumped out, delete the symbolic breakpoint by right-clicking the breakpoint in the breakpoint navigator and selecting **Delete Breakpoint**.

In addition to symbolic breakpoints, Xcode also supports several types of error breakpoints. One of these is the **Exception Breakpoint**. Sometimes, something goes wrong in your program and it just simply crashes. When this happens, your first reaction to this should be to enable an exception breakpoint, which will fire every time an exception is thrown. Xcode will show you the offending line, which greatly aids in hunting down the culprit responsible for the crash.

Finally, there is the **Swift Error Breakpoint**, which stops any time Swift throws an error by essentially creating a breakpoint on the `swift_willThrow` method. This is a great option to use if you're working with any APIs that can be error-prone, as it lets you diagnose the situation quickly without making false assumptions about the correctness of your code.

LLDB breakpoint syntax

Now that you've had a crash course in using the IDE debugging features of Xcode, it's time to learn how to create breakpoints through the LLDB console. In order to create useful breakpoints, you need to learn how to query what you're looking for.

The `image` command is an excellent tool to help introspect details that will be vital for setting breakpoints.

There are two configurations you'll use in this book for code hunting. The first is the following:

```
(lldb) image lookup -n "-[UIViewController viewDidLoad]"
```

This command dumps the implementation address (the offset address of where this method is located within the framework's binary) of the function for `-[UIViewController viewDidLoad]`. The `-n` argument tells LLDB to look up either a symbol or function name. The output will be similar to below:

```
1 match found in /Applications/Xcode.app/Contents/Developer/Platforms/
iPhoneOS.platform/Developer/Library/CoreSimulator/Profiles/Runtimes/
iOS.simruntime/Contents/Resources/RuntimeRoot/System/Library/
PrivateFrameworks/UIKitCore.framework/UIKitCore:
        Address: UIKitCore[0x0000000000ac813c] (UIKitCore.__TEXT.__text +
11295452)
        Summary: UIKitCore`-[UIViewController viewDidLoad]
```

You can tell that this is iOS 12 due to the location of where the `-[UIViewController viewDidLoad]` method is located. In previous iOS versions this method was located in `UIKit`, but has now since moved to `UIKitCore` likely due to the macOS/iOS unification process, unofficially referred to as **Marzipan**.

Another useful, similar command is this:

```
(lldb) image lookup -rn test
```

This does a case-sensitive regex lookup for the word `"test"`. If the lowercase word `"test"` is found anywhere, in any function, in any of the modules (i.e. UIKit, Foundation, Core Data, etc) loaded in the current executable (that are not stripped out of a release builds... more on that later), this command will spit out the results.

> **Note**: Use the `-n` argument when you want exact matches (with quotes around your query if it contains spaces) and use the `-rn` arguments to do a regex search. The `-n` only command helps figure out the exact parameters to match a breakpoint, especially when dealing with Swift, while the `-rn` argument option will be heavily favored in this book since a smart regex can eliminate quite a bit of typing — as you'll soon find out.

Objective-C properties

Learning how to query loaded code is essential for learning how to create breakpoints on that code. Both Objective-C and Swift have specific property signatures when they're created by the compiler, which results in different querying strategies when looking for code.

For example, the following Objective-C class is declared in the Signals project:

```
@interface TestClass : NSObject
@property (nonatomic, strong) NSString *name;
@end
```

The compiler will generate code for both the setter and getter of the property `name`. The getter will look like the following:

```
-[TestClass name]
```

...while the setter would look like this:

```
-[TestClass setName:]
```

Build and run the app, then pause the debugger. Next, verify these methods do exist by typing the following into LLDB:

```
(lldb) image lookup -n "-[TestClass name]"
```

In the console output, you'll get something similar to the following:

```
1 match found in /Users/derekselander/Library/Developer/Xcode/
DerivedData/Signals-atqcdyprrotlrvdanihoufkwzyqh/Build/Products/Debug-
iphonesimulator/Signals.app/Signals:
        Address: Signals[0x0000000100002150] (Signals.__TEXT.__text + 0)
        Summary: Signals`-[TestClass name] at TestClass.h:28
```

LLDB will dump information about the function included in the executable. The output may look scary, but there are some good tidbits here.

> **Note**: The `image lookup` command can produce a lot of output that can be pretty hard on the eyes when a query matches a lot of code. In Chapter 26, "SB Examples, Improved Lookup", you'll build a cleaner alternative to LLDB's `image lookup` command to save your eyes from looking at too much output.

The console output tells you LLDB was able to find out this function was implemented in the Signals executable, at an offset of `0x0000000100002150` in the `__TEXT` segment of the `__text` section to be exact (don't worry if that didn't make any sense, you'll learn all about that later on in the book). LLDB was also able to tell that this method was declared on line 28 in `TestClass.h`.

You can check for the setter as well, like so:

```
(lldb) image lookup -n "-[TestClass setName:]"
```

You'll get output similar to the previous command, this time showing the implementation address and of the setter's declaration for `name`.

Objective-C properties and dot notation

Something that is often misleading to entry level Objective-C (or Swift only) developers is the Objective-C dot notation syntax for properties.

Objective-C dot notation is a somewhat controversial compiler feature that allows properties to use a shorthand getter or setter.

Consider the following:

```
TestClass *a = [[TestClass alloc] init];

// Both equivalent for setters
[a setName:@"hello, world"];
a.name = @"hello, world";

// Both equivalent for getters
NSString *b;
b = [a name]; // b = @"hello, world"
b = a.name;   // b = @"hello, world"
```

In the above example, the `-[TestClass setName:]` method is called twice, even with the dot notation. The same can be said for the getter, `-[TestClass name]`. This is important to know if you're dealing with Objective-C code and trying to create breakpoints on the setters and getters of properties with dot notation.

Swift properties

The syntax for a property is much different in Swift. Take a look at the code in **SwiftTestClass.swift** which contains the following:

```
class SwiftTestClass: NSObject {
  var name: String!
}
```

Make sure the **Signals** project is running and paused in LLDB. Feel free to clear the LLDB console by typing **Command + K** in the debug window to start fresh.

In the LLDB console, type the following:

```
(lldb) image lookup -rn Signals.SwiftTestClass.name.setter
```

You'll get output similar to below:

```
1 match found in /Users/derekselander/Library/Developer/Xcode/
DerivedData/Signals-atqcdyprrotlrvdanihoufkwzyqh/Build/Products/Debug-
iphonesimulator/Signals.app/Signals:
        Address: Signals[0x000000010000bd50] (Signals.__TEXT.__text +
39936)
        Summary: Signals`Signals.SwiftTestClass.name.setter :
Swift.Optional<Swift.String> at SwiftTestClass.swift:28
```

Hunt for the information after the word **Summary** in the output. There are a couple of interesting things to note here.

Do you see how long the function name is!? This whole thing needs to be typed out for *one* valid Swift breakpoint! If you wanted to set a breakpoint on this setter, you'd have to type the following:

```
(lldb) b Signals.SwiftTestClass.name.setter :
Swift.Optional<Swift.String>
```

Using regular expressions is an attractive alternative to typing out this monstrosity.

Apart from the length of the Swift function name you produced, note how the Swift property is formed. The function signature containing the property `name` has the word `setter` immediately following the property. Perhaps the same convention works for the getter method as well?

Search for the `SwiftTestClass` setter and getter for the `name` property, at the same time, using the following regular expression query:

```
(lldb) image lookup -rn Signals.SwiftTestClass.name
```

This uses a regex query to dump everything that contains the phrase `Signals.SwiftTestClass.name`.

Since this is a regular expression, the periods (`.`) are evaluated as wildcards, which in turn matches periods in the actual function signatures.

You'll get a fair bit of output, but hone in every time you see the word **Summary** in the console ouput. You'll find the output matches the getter, (`Signals.SwiftTestClass.name.getter`) the setter, (`Signals.SwiftTestClass.name.setter`), as well as two methods containing `materializeForSet`, helper methods for Swift constructors.

There's a pattern for the function names for Swift properties:

`ModuleName.Classname.PropertyName.(getter|setter)`

The ability to dump methods, find a pattern, and narrow your search scope is a great way to uncover the Swift/Objective-C language internals as you work to create smart breakpoints in your code.

Finally... creating breakpoints

Now you know how to query the existence of functions and methods in your code, it's time to start creating breakpoints on them.

If you already have the Signals app running, stop and restart the application, then press the pause button to stop the application and bring up the LLDB console.

There are several different ways to create breakpoints. The most basic way is to simply type the letter **b** followed by the name of your breakpoint. This is fairly easy in Objective-C and C, since the names are short and easy to type (e.g. `-[NSObject init]` or `-[UIView setAlpha:]`). They're quite tricky to type in C++ and Swift, since the compiler turns your methods into symbols with rather long names.

Since UIKit is primarily Objective-C (at the time of this writing at least!), create a breakpoint using the `b` argument, like so:

```
(lldb) b -[UIViewController viewDidLoad]
```

You'll see the following output:

```
Breakpoint 1: where = UIKitCore`-[UIViewController viewDidLoad], address
= 0x0000000114a4a13c
```

When you create a valid breakpoint, the console will spit out some information about that breakpoint. In this particular case, the breakpoint was created as **Breakpoint 1** since this was the first breakpoint in this particular debugging session. As you create more breakpoints, this breakpoint ID will increment.

Resume the debugger. Once you've resumed execution, a new `SIGSTOP` signal will be displayed. Tap on the cell to bring up the detail `UIViewController`. The program should pause when `viewDidLoad` of the detail view controller is called.

> **Note**: Like a lot of shorthand commands, `b` is an abbreviation for another, longer LLDB command. Run the `help` with the `b` command to figure out the actual command yourself and learn all the cool tricks `b` can do under the hood.

In addition to the `b` command, there's another longer `breakpoint set` command, which has a slew of options available. You'll explore these options over the next couple of sections. Many of the commands will stem from various options of the `breakpoint set` command.

Regex breakpoints and scope

Another extremely powerful command is the regular expression breakpoint, `rbreak`, which is an abbreviation for `breakpoint set -r %1`. You can quickly create many breakpoints using smart regular expressions to stop wherever you want.

Going back to the previous example with the egregiously long Swift property function names, instead of typing:

```
(lldb) b Signals.SwiftTestClass.name.setter :
Swift.Optional<Swift.String>
```

You can simply type:

```
(lldb) rb SwiftTestClass.name.setter
```

The `rb` command will get expanded out to `rbreak` (provided you don't have any other LLDB commands that begin with "rb"). This will create a breakpoint on the setter property of `name` in `SwiftTestClass`

To be even more brief, you could simply use the following:

```
(lldb) rb name\.setter
```

This will produce a breakpoint on anything that contains the phrase `name.setter`. This will work if you know you don't have any other Swift properties called `name` within your project; otherwise you'll create multiple breakpoints for each class that contains a "name" property that has a setter.

Let's up the complexity of these regular expressions.

Create a breakpoint on every Objective-C instance method of `UIViewController`. Type the following into your LLDB session:

```
(lldb) rb '\-\[UIViewController\ '
```

The ugly back slashes are escape characters to indicate you want the literal character to be in the regular expression search. As a result, this query breaks on every method containing the string `-[UIViewController` followed by a space.

But wait… what about Objective-C categories? They take on the form of `(-|+) [ClassName(categoryName) method]`. You'll have to rewrite the regular expression to include categories as well.

Type the following into your LLDB session and when prompted type `y` to confirm:

```
(lldb) breakpoint delete
```

This command deletes all the breakpoints you have set.

Next, type the following:

```
(lldb) rb '\-\[UIViewController(\(\w+\))?\ '
```

This provides an optional parenthesis with one or more alphanumeric characters followed by a space, after `UIViewController` in the breakpoint.

Regex breakpoints let you capture a wide variety of breakpoints with a single expression.

You can limit the scope of your breakpoints to a certain file, using the `–f` option. For example, you could type the following:

```
(lldb) rb . –f DetailViewController.swift
```

This would be useful if you were debugging **DetailViewController.swift**. It would set a breakpoint on all the property getters/setters, blocks/closures, extensions/categories, and functions/methods in this file. `–f` is known as a **scope limitation**.

If you were completely crazy and a fan of pain (the doctors call that masochistic?), you could omit the scope limitation and simply do this:

```
(lldb) rb .
```

This will create a breakpoint on everything… Yes, everything! This will create breakpoints on all the code in the **Signals** project, all the code in UIKit as well as Foundation, all the event run loop code that gets fired at (hopefully) 60 hertz — everything. As a result, expect to type `continue` in the debugger a fair bit if you execute this.

There are other ways to limit the scope of your searches. You can limit to a single library using the `–s` option:

```
(lldb) rb . –s Commons
```

This would set a breakpoint on everything within the `Commons` library, which is a dynamic library contained within the **Signals** project.

This is not limited to your code; you can use the same tactic to create a breakpoint on every function in UIKitCore, like so:

```
(lldb) rb . –s UIKitCore
```

Even *that* is still a little crazy. There are a lot of methods — around 86,760 UIKitCore methods in iOS 12.0. How about only stopping on the first method in UIKitCore you hit, and simply continue? The `–o` option offers a solution for this. It creates what is known as a "one-shot" breakpoint. When these breakpoints hit, the breakpoint is deleted. So it'll only ever hit once.

To see this in action, type the following in your LLDB session:

```
(lldb) breakpoint delete
(lldb) rb . -s UIKitCore -o 1
```

> **Note**: Be patient while your computer executes this command, as LLDB has to create a lot of breakpoints. Also make sure you are using the Simulator, or else you'll wait for a very long time!

Next, continue the debugger, and click on a cell in the table view. The debugger stops on the first UIKitCore method this action calls. Finally, continue the debugger, and the breakpoint will no longer fire.

Other cool breakpoint options

The `-L` option lets you filter by source language. So, if you wanted to only go after Swift code in the `Commons` module of the Signals application, you could do the following:

```
(lldb) breakpoint set -L swift -r . -s Commons
```

This would set a breakpoint on every Swift method within the Commons module.

What if you wanted to go after something interesting around a Swift `if let` but totally forgot where in your application it is? You can use **source regex breakpoints** to help figure locations of interest! Like so:

```
(lldb) breakpoint set -A -p "if let"
```

This will create a breakpoint on every source code location that contains `if let`. You can of course get waaaaaay more fancy since the `-p` takes a regular expression breakpoint to go after complicated expressions. The `-A` option says to search in all source files known to the project.

If you wanted to filter the above breakpoint query to only `MasterViewController.swift` and `DetailViewController.swift`, you could do the following:

```
(lldb) breakpoint set -p "if let" -f MasterViewController.swift -f
DetailViewController.swift
```

Notice how the `-A` has gone, and how each `-f` will let you specify a filename. I am lazy, so I'll usually default to `-A` to give me all files and drill in from there.

Finally, you can also filter by a specific module as well. If you wanted to create a breakpoint for "if let" for anything in the `Signals` executable (while ignoring other frameworks like Commons), you could do this:

```
(lldb) breakpoint set -p "if let" -s Signals -A
```

This will grab all source files (`-A`), but filter those to only the ones that belong to the `Signals` executable (with the `-s Signals` option).

One more cool breakpoint option example? OK, you talked me into it. You will make a breakpoint which prints the `UIViewController` whenever `viewDidLoad` gets hit, but you'll do it via LLDB console instead of the Symbolic breakpoint window. Then, you'll export this breakpoint to a file so you can show how cool you are to your coworkers by using the `breakpoint read` and `breakpoint write` commands!

First off, delete all breakpoints:

```
(lldb) breakpoint delete
```

Now create the following (complex!) breakpoint:

```
(lldb) breakpoint set -n "-[UIViewController viewDidLoad]" -C "po $arg1"
-G1
```

Make sure to use a capitol `-C`, since LLDB's `-c` performs a different option!

This says to create a breakpoint on `-[UIViewController viewDidLoad]`, then execute the (C)ommand "po $arg1", which prints out the instance of the `UIViewController`. From there, the `-G1` option tells the breakpoint to automatically continue after executing the command.

Verify the console displays the expected information by triggering a `viewDidLoad` by tapping on one of the `UITableViewCell`s containing a Unix signal.

Now, how can you send this to a coworker? In LLDB, type the following:

```
(lldb) breakpoint write -f /tmp/br.json
```

This will write all the breakpoints in your session to the `/tmp/br.json` file. You can specify a single breakpoint or list of breakpoints by breakpoint ID, but that's for you to determine via the help documentation on your own time.

You can verify the breakpoint data either in Terminal or via LLDB by using the **platform shell** command to breakout into using Terminal.

Use the **cat** Terminal command to display the breakpoint data.

```
(lldb) platform shell cat /tmp/br.json
```

This means that you can send over this file to your coworker and have her open it via the **breakpoint read** command.

To simulate this, delete all breakpoints again.

```
(lldb) breakpoint delete
```

You will now have a clean debugging session with no breakpoints.

Now, re-import your custom breakpoint command:

```
(lldb) breakpoint read -f /tmp/br.json
```

Once again, if you were to trigger the `UIViewController`'s `viewDidLoad` method, the instance will be printed out due to your custom breakpoint logic! Using these commands, you can easily send and recieve LLDB breakpoint commands to help replicate a hard to catch bug!

Modifying and removing breakpoints

Now that you have a basic understanding of how to create these breakpoints, you might be wondering how you can alter them. What if you found the object you were interested in and wanted to delete the breakpoint, or temporarily disable it? What if you need to modify the breakpoint to perform a specific action next time it triggers?

First, you'll need to discover how to uniquely identify a breakpoint or a group of breakpoints.

Build and run the app to get a clean LLDB session. Next, pause the debugger and type the following into the LLDB session:

```
(lldb) b main
```

The output will look something similar to the following:

```
Breakpoint 1: 70 locations.
```

This creates a breakpoint with 70 locations, matching the function `"main"` in various modules.

In this case, the breakpoint ID is 1, because it's the first breakpoint you created in this session. To see details about this breakpoint you can use the `breakpoint list` subcommand. Type the following:

```
(lldb) breakpoint list 1
```

The output will look similar to the truncated output below:

```
1: name = 'main', locations = 70, resolved = 70, hit count = 0
  1.1: where = Signals`main at AppDelegate.swift, address =
0x00000001098b1520, resolved, hit count = 0
  1.2: where = Foundation`-[NSThread main], address = 0x0000000109bfa9e3,
resolved, hit count = 0
  1.3: where = Foundation`-[NSBlockOperation main], address =
0x0000000109c077d6, resolved, hit count = 0
  1.4: where = Foundation`-[NSFilesystemItemRemoveOperation main],
address = 0x0000000109c40e99, resolved, hit count = 0
  1.5: where = Foundation`-[NSFilesystemItemMoveOperation main], address
= 0x0000000109c419ee, resolved, hit count = 0
  1.6: where = Foundation`-[NSInvocationOperation main], address =
0x0000000109c6aee4, resolved, hit count = 0
  1.7: where = Foundation`-[NSDirectoryTraversalOperation main], address
= 0x0000000109caefa6, resolved, hit count = 0
  1.8: where = Foundation`-[NSOperation main], address =
0x0000000109cfd5e3, resolved, hit count = 0
  1.9: where = Foundation`-
[_NSFileAccessAsynchronousProcessAssertionOperation main], address =
0x0000000109d55ca9, resolved, hit count = 0
  1.10: where = UIKit`-[_UIFocusFastScrollingTest main], address =
0x000000010b216598, resolved, hit count = 0
  1.11: where = UIKit`-[UIStatusBarServerThread main], address =
0x000000010b651e97, resolved, hit count = 0
  1.12: where = UIKit`-[_UIDocumentActivityDownloadOperation main],
address = 0x000000010b74f718, resolved, hit count = 0
```

This shows the details of that breakpoint, including all locations that include the word `"main"`.

A cleaner way to view this is to type the following:

```
(lldb) breakpoint list 1 -b
```

This will give you output that is a little easier on the visual senses. If you have a breakpoint ID that encapsulates a lot of breakpoints, this brief flag is a good solution.

If you want to query all the breakpoints in your LLDB session, simply omit the ID like so:

```
(lldb) breakpoint list
```

You can also specify multiple breakpoint IDs and ranges:

```
(lldb) breakpoint list 1 3
(lldb) breakpoint list 1-3
```

Using `breakpoint delete` to delete all breakpoints is a bit heavy-handed. You can simply use the same ID pattern used in the breakpoint list command to delete a set.

You can delete a single breakpoint by specifying the ID like so:

```
(lldb) breakpoint delete 1
```

However, your breakpoint for `"main"` had 70 locations (maybe more or less depending on the iOS version). You can also delete a single location, like so:

```
(lldb) breakpoint delete 1.1
```

This would delete the first sub-breakpoint of breakpoint 1, which results in only one `main` function breakpoint removed while keeping the remaining `main` breakpoints active.

Where to go from here?

You've covered a lot in this chapter. Breakpoints are a big topic and mastering the art of quickly finding an item of interest is essential to becoming a debugging expert. You've also started exploring function searching using regular expressions. Now would be a great time to brush up on regular expression syntax, as you'll be using lots of regular expressions in the rest of this book.

Check out https://docs.python.org/2/library/re.html to learn (or relearn) regular expressions. Try figuring out how to make a case-insensitive breakpoint query.

You've only begun to discover how the compiler generates functions in Objective-C and Swift. Try to figure out the syntax for stopping on Objective-C blocks or Swift closures. Once you've done that, try to design a breakpoint that only stops on Objective-C blocks within the Commons framework of the **Signals** project. These are regex skills you'll need in the future to construct ever more complicated breakpoints.

Chapter 5: Expression

Now that you've learned how to set breakpoints so the debugger will stop in your code, it's time to get useful information out of whatever software you're debugging.

You'll often want to inspect instance variables of objects. But, did you know you can even execute arbitrary code through LLDB? What's more, by using the Swift/Objective-C APIs you can declare, initialize, and inject code all on the fly to help aid in your understanding of the program.

In this chapter you'll learn about the **`expression`** command. This allows you to execute arbitrary code in the debugger.

Formatting p and po

You might be familiar with the go-to debugging command, **`po`**. `po` is often used in Swift & Objective-C code to print out an item of interest. This could be an instance variable in an object, a local reference to an object, or a register, as you've seen earlier in this book. It could even be an arbitrary memory reference — so long as there's an object at that address!

If you do a quick `help po` in the LLDB console, you'll find `po` is actually a shorthand expression for `expression -O --`. The `-O` arugment is used to print the object's description.

`po`'s often overlooked sibling, **`p`**, is another abbreviation with the `-O` option omitted, resulting in `expression --`. The format of what `p` will print out is more dependent on the **LLDB type system**. LLDB's type formatting of values helps determine its output and is fully customizable (as you'll see in a second).

It's time to learn how the `p` and `po` commands get their content. You'll continue using the Signals project for this chapter.

Start by opening the Signals project in Xcode. Next, open **MasterViewController.swift** and add the following code above `viewDidLoad()`:

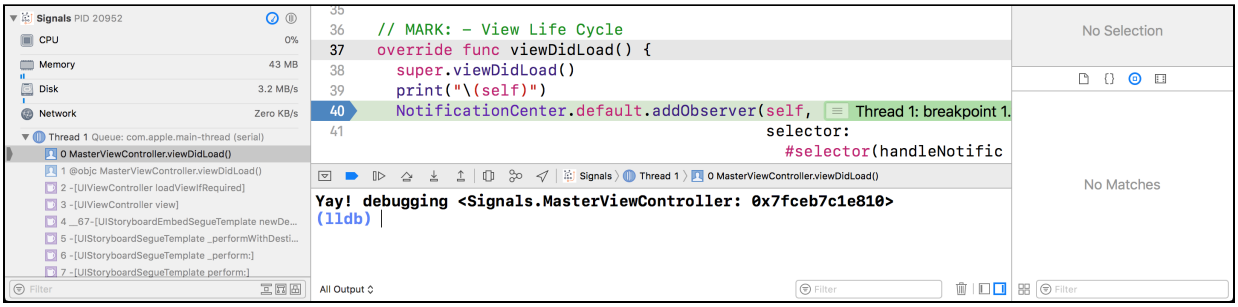
```
override var description: String {
  return "Yay! debugging " + super.description
}
```

In `viewDidLoad`, add the following line of code below `super.viewDidLoad()`:

```
print("\(self)")
```

Now, put a breakpoint just after the print method you created in the `viewDidLoad()` of **MasterViewController.swift**. Do this using the Xcode GUI breakpoint side panel.

Build and run the application.



Once the Signals project stops at `viewDidLoad()`, type the following into the LLDB console:

```
(lldb) po self
```

You'll get output similar to the following:

```
Yay! debugging <Signals.MasterViewController: 0x7f8a0ac06b70>
```

Take note of the output of the `print` statement and how it matches the `po self` you just executed in the debugger.

You can also take it a step further. `NSObject` has an additional method description used for debugging called **`debugDescription`**. Add the following below your `description` variable definition:

```
override var debugDescription: String {
  return "debugDescription: " + super.debugDescription
}
```

Build and run the application. When the debugger stops at the breakpoint, print `self` again:

```
(lldb) po self
```

The output from the LLDB console will look similar to the following:

```
debugDescription: Yay! debugging <Signals.MasterViewController:
0x7fb71fd04080>
```

Notice how the `po self` and the output of `self` from the `print` command now differ, since you implemented `debugDescription`. When you print an object from LLDB, it's `debugDescription` that gets called, rather than `description`. Neat!

As you can see, having a `description` or `debugDescription` when working with an `NSObject` class or subclass will influence the output of `po`.

So which objects override these description methods? You can easily hunt down which objects override these methods using the `image lookup` command with a smart regex query. Your learnings from previous chapters are already coming in handy!

For example, if you wanted to know all the Objective-C classes that override `debugDescription`, you can simply query all the methods by typing:

```
(lldb) image lookup -rn '\ debugDescription\]'
```

Based upon the output, it seems the authors of the Foundation framework have added the `debugDescription` to a lot of foundation types (i.e. `NSArray`), to make our debugging lives easier. In addition, they're also private classes that have overridden `debugDescription` methods as well.

You may notice one of them in the listing is `CALayer`. Let's take a look at the difference between `description` and `debugDescription` in `CALayer`.

In your LLDB console, type the following:

```
(lldb) po self.view!.layer.description
```

You'll see something similar to the following:

```
"<CALayer: 0x600002e9eb00>"
```

That's a little boring. Now type the following:

```
(lldb) po self.view!.layer
```

You'll see something similar to the following:

```
<CALayer:0x600002e9eb00; position = CGPoint (187.5 406); bounds = CGRect
(0 0; 375 812); delegate = <UITableView: 0x7fc25c01c600; frame = (0 0;
375 812); clipsToBounds = YES; autoresize = W+H; gestureRecognizers =
<NSArray: 0x6000020cf240>; layer = <CALayer: 0x600002e9eb00>;
contentOffset: {0, 0}; contentSize: {0, 0}; adjustedContentInset: {0, 0,
0, 0}>; sublayers = (<CALayer: 0x600002e9f2a0>, <CALayer:
0x600002e9f340>); masksToBounds = YES; allowsGroupOpacity = YES;
backgroundColor = <CGColor 0x600000a88b40> [<CGColorSpace 0x600000a83b40>
(kCGColorSpaceICCBased; kCGColorSpaceModelRGB; sRGB IEC61966-2.1;
extended range)] ( 0.980392 0.980392 0.980392 1 ); _uikit_viewPointer =
<UITableView: 0x7fc25c01c600; frame = (0 0; 375 812); clipsToBounds =
YES; autoresize = W+H; gestureRecognizers = <NSArray: 0x6000020cf240>;
layer = <CALayer: 0x600002e9eb00>; contentOffset: {0, 0}; contentSize:
{0, 0}; adjustedContentInset: {0, 0, 0, 0}>>
```

That's much more interesting — and much more useful! Obviously the developers of Core Animation decided the plain `description` should be just the object reference, but if you're in the debugger, you'll want to see more information. It's unclear exactly why they did this. It might be some of the information in the debug description is expensive to calculate, so they only want to do it when absolutely necessary.

Next, while you're still stopped in the debugger (and if not, get back to the `viewDidLoad()` breakpoint), execute the `p` command on `self`, like so:

```
(lldb) p self
```

You'll get something similar to the following:

```
(Signals.MasterViewController) $R2 = 0x00007fc25b80e6e0 {
  UIKit.UITableViewController = {
    baseUIViewController@0 = <extracting data from value failed>

    _tableViewStyle = 0
    _keyboardSupport = nil
    _staticDataSource = nil
    _filteredDataSource = 0x00006000020ceee0
    _filteredDataType = 0
  }
  detailViewController = nil
}
```

This might look scary, but let's break it down.

First, LLDB spits out the class name of `self`. In this case, `Signals.MasterViewController`.

Next follows a reference you can use to refer to this object from now on within your LLDB session. In the example above, it's `$R2`. Yours will vary as this is a number LLDB increments as you use LLDB.

This reference is useful if you ever want to get back to this object later in the session, perhaps when you're in a different scope and `self` is no longer the same object. In that case, you can refer back to this object as `$R2`. To see how, type the following:

```
(lldb) p $R2
```

You'll see the same information printed out again. You'll learn more about these LLDB variables later in this chapter.

After the LLDB variable name is the address to this object, followed by some output specific to this type of class. In this case, it shows the details relevant to `UITableViewController`, which is the superclass of `MasterViewController`, followed by the `detailViewController` instance variable.

As you can see, the meat of the output of the `p` command is different to the `po` command. The output of `p` is dependent upon **type formatting**: internal data structures the LLDB authors have added to every (noteworthy) data structure in Objective-C, Swift, and other languages. It's important to note the formatting for Swift is under active development with every Xcode release, so the output of `p` for `MasterViewController` might be different for you.

Since these type formatters are held by LLDB, you have the power to change them if you so desire. In your LLDB session, type the following:

```
(lldb) type summary add Signals.MasterViewController --summary-string
"Wahoo!"
```

You've now told LLDB you just want to return the static string, `"Wahoo!"`, whenever you print out an instance of the MasterViewController class. The `Signals` prefix is essential for Swift classes since Swift includes the module in the classname to prevent namespace collisions. Try printing out `self` now, like so:

```
(lldb) p self
```

The output should look similar to the following:

```
(lldb) (Signals.MasterViewController) $R3 = 0x00007fb71fd04080 Wahoo!
```

This formatting will be remembered by LLDB across app launches, so be sure to remove it when you're done playing with the `p` command. Remove yours from your LLDB session like so:

```
(lldb) type summary clear
```

Typing `p self` will now go back to the default implementation created by the LLDB formatting authors.

Swift vs Objective-C debugging contexts

It's important to note there are two debugging contexts when debugging your program: a non-Swift debugging context and a Swift context. By default, when you stop in Objective-C code, LLDB will use the non-Swift (Objective-C) debugging context, while if you're stopped in Swift code, LLDB will use the Swift context. Sounds logical, right?

If you stop the debugger out of the blue (for example, if you press the process pause button in Xcode), LLDB will choose the Objective-C context by default.

Make sure the GUI Swift breakpoint you've created in the previous section is still enabled and build and run the app. When the breakpoint hits, type the following into your LLDB session:

```
(lldb) po [UIApplication sharedApplication]
```

LLDB will throw a cranky error at you:

```
error: <EXPR>:3:16: error: expected ',' separator
[UIApplication sharedApplication]
               ^
              ,
```

You've stopped in Swift code, so you're in the Swift context. But you're trying to execute Objective-C code. That won't work. Similarly, in the Objective-C context, doing a `po` on a Swift object will not work.

You can force the expression to be used in the Objective-C context with the `-l` option to select the language. However, since the `po` expression is mapped to `expression -O --`, you'll be unable to use the `po` command since the arguments you provide come *after* the `--`, which means you'll have to type out the `expression`. In LLDB, type the following:

```
(lldb) expression -l objc -O -- [UIApplication sharedApplication]
```

Here you've told LLDB to use the `objc` language for Objective-C. You can also use `objc++` for Objective-C++ if necessary.

LLDB will spit out the reference to the shared application. Try the same thing in Swift. Since you're already stopped in the Swift context, try to print the UIApplication reference using Swift syntax, like so:

```
(lldb) po UIApplication.shared
```

You'll get the same output as you did printing with the Objective-C context. Resume the program, by typing `continue`, then pause the Signals application out of the blue.

From there, press the up arrow to bring up the same Swift command you just executed and see what happens:

```
(lldb) po UIApplication.shared
```

Again, LLDB will be cranky:

```
error: property 'shared' not found on object of type 'UIApplication'
```

Remember, stopping out of the blue will put LLDB in the Objective-C context. That's why you're getting this error when trying to execute Swift code.

You should always be aware of the language in which you are currently paused in the debugger.

User defined variables

As you saw earlier, LLDB will automatically create local variables on your behalf when printing out objects. You can create your own variables as well.

Remove all the breakpoints from the program and build and run the app. Stop the debugger out of the blue so it defaults to the Objective-C context. From there type:

```
(lldb) po id test = [NSObject new]
```

LLDB will execute this code, which creates a new `NSObject` and stores it to the `test` variable. Now, print the `test` variable in the console:

```
(lldb) po test
```

You'll get an error like the following:

```
error: use of undeclared identifier 'test'
```

This is because you need to prepend variables you want LLDB to remember with the **`$`** character.

Declare `test` again with the `$` in front:

```
(lldb) po id $test = [NSObject new]
(lldb) po $test
<NSObject: 0x60000001d190>
```

This variable was created in the Objective-C object. But what happens if you try to access this from the Swift context? Try it, by typing the following:

```
(lldb) expression -l swift -O -- $test
```

So far so good. Now try executing a Swift-styled method on this Objective-C class.

```
(lldb) expression -l swift -O -- $test.description
```

You'll get an error like this:

```
error: <EXPR>:3:1: error: use of unresolved identifier '$test'
$test.description
^~~~
```

If you create an LLDB variable in the Objective-C context, then move to the Swift context, don't expect everything to "just work." This is an area under active development and the bridging between Objective-C and Swift through LLDB will likely see improvements over time.

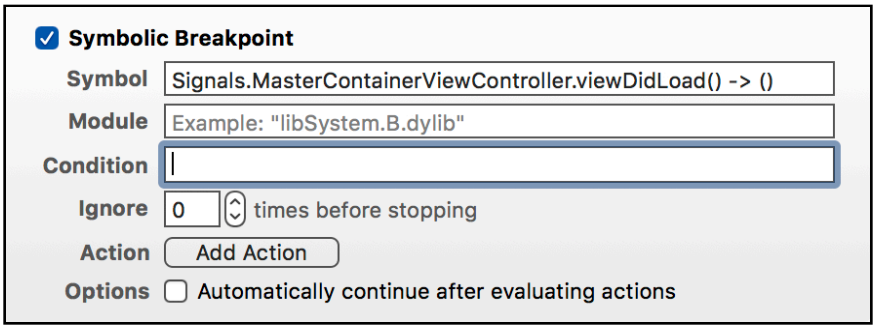
So how could creating references in LLDB actually be used in a real life situation? You can grab the reference to an object and execute (as well as debug!) arbitrary methods of your choosing. To see this in action, create a symbolic breakpoint on MasterViewController's parent view controller, **MasterContainerViewController** using an Xcode symbolic breakpoint for MasterContainerViewController's `viewDidLoad`.

In the Symbol section, type the following:

```
Signals.MasterContainerViewController.viewDidLoad() -> ()
```

Be aware of the spaces for the parameters and parameter return type, otherwise the breakpoint will not work.

Your breakpoint should look like the following:



Build and run the app. Xcode will now break on `MasterContainerViewController.viewDidLoad()`. From there, type the following:

```
(lldb) p self
```

Since this is the first argument you executed in the Swift debugging context, LLDB will create the variable, **`$R0`**. Resume execution of the program by typing `continue` in LLDB.

Now you don't have a reference to the instance of `MasterContainerViewController` through the use of `self` since the execution has left `viewDidLoad()` and moved on to bigger and better run loop events.

Oh, wait, you still have that `$R0` variable! You can now reference `MasterContainerViewController` and even execute arbitrary methods to help debug your code.

Pause the app in the debugger manually, then type the following:

```
(lldb) po $R0.title
```

Unfortunately, you get:

```
error: use of undeclared identifier '$R0'
```

You stopped the debugger out of the blue! Remember, LLDB will default to Objective-C; you'll need to use the `-l` option to stay in the Swift context:

```
(lldb) expression -l swift -- $R0.title
```

The output will be similar to the following, you might have a different **R** number:

```
(String?) $R1 = "Quarterback"
```

Of course, this is the title of the view controller, shown in the navigation bar.

Now, type the following:

```
(lldb) expression -l swift -- $R0.title = "💩💩💩💩💩"
```

Resume the app by typing `continue` or pressing the play button in Xcode.

> **Note**: To quickly access a poop emoji on your macOS machine, hold down `⌘ + ^ + space`. From there, you can easily hunt down the correct emoji by searching for the phrase "poop."

It's the small things in life you cherish!



As you can see, you can easily manipulate variables to your will.

In addition, you can also create a breakpoint on code, execute the code, and cause the breakpoint to be hit. This can be useful if you're in the middle of debugging something and want to step through a function with certain inputs to see how it operates.

For example, you still have the symbolic breakpoint in `viewDidLoad()`, so try executing that method to inspect the code. Pause execution of the program, then type:

```
(lldb) expression -l swift -O -- $R0.viewDidLoad()
```

Nothing happened. The breakpoint didn't hit. What gives? In fact, `MasterContainerViewController` *did* execute the method, but by default, LLDB will ignore any breakpoints when executing commands. You can disable this option with the `-i` option.

Type the following into your LLDB session:

```
(lldb) expression -l swift -O -i 0 -- $R0.viewDidLoad()
```

LLDB will now break on the `viewDidLoad()` symbolic breakpoint you created earlier. This tactic is a great way to test the logic of methods. For example, you can implement test-driven debugging, by giving a function different parameters to see how it handles different input.

Type formatting

One of the nice options LLDB has is the ability to format the output of basic data types. This makes LLDB a great tool to learn how the compiler formats basic C types. This is a must to know when you're exploring the assembly section, which you'll do later in this book.

First, remove the previous symbolic breakpoint. Next, build and run the app and finally pause the debugger out of the blue to make sure you're in the Objective-C context.

Type the following into your LLDB session:

```
(lldb) expression -G x -- 10
```

This `-G` option tells LLDB what format you want the output in. The G stands for **GDB format**. If you're not aware, GDB is the debugger that preceded LLDB. This therefore is saying whatever you specify is a GDB format specifier. In this case, `x` is used which indicates hexadecimal.

You'll see the following output:

```
(int) $0 = 0x0000000a
```

This is decimal 10 printed as hexadecimal. Wow!

But wait! There's more! LLDB lets you format types using a neat shorthand syntax. Type the following:

```
(lldb) p/x 10
```

You'll see the same output as before. But that's a lot less typing!

This is great for learning the representations behind C datatypes. For example, what's the binary representation of the integer 10?

```
(lldb) p/t 10
```

The `/t` specifies binary format. You'll see what decimal 10 looks like in binary. This can be particularly useful when you're dealing with a bit field for example, to double check what fields will be set for a given number.

What about negative 10?

```
(lldb) p/t -10
```

Decimal 10 in two's complement. Neat!

What about the floating point binary representation of 10.0?

```
(lldb) p/t 10.0
```

That could come in handy!

How about the ASCII value of the character 'D'?

```
(lldb) p/d 'D'
```

Ah so 'D' is 68! The `/d` specifies decimal format.

Finally, what is the acronym hidden behind this integer?

```
(lldb) p/c 1430672467
```

The `/c` specifies `char` format. It takes the number in binary, splits into 8 bit (1 byte) chunks, and converts each chunk into an ASCII character. In this case, it's a 4 character code (FourCC), saying `STFU`. Hey! Be nice now!

The full list of output formats is as follows (taken from https://sourceware.org/gdb/onlinedocs/gdb/Output-Formats.html):

- `x`: hexadecimal
- `d`: decimal
- `u`: unsigned decimal
- `o`: octal
- `t`: binary
- `a`: address
- `c`: character constant
- `f`: float
- `s`: string

If these formats aren't enough for you, you can use LLDB's extra formatters, although you'll be unable to use the GDB formatting syntax.

LLDB's formatters can be used like this:

```
(lldb) expression -f Y -- 1430672467
```

This gives you the following output:

```
(int) $0 = 53 54 46 55              STFU
```

This explains the FourCC code from earlier!

LLDB has the following formatters (taken from http://lldb.llvm.org/varformats.html):

- `B`: boolean
- `b`: binary
- `y`: bytes
- `Y`: bytes with ASCII
- `c`: character
- `C`: printable character
- `F`: complex float
- `s`: c-string
- `i`: decimal
- `E`: enumeration
- `x`: hex
- `f`: float
- `o`: octal
- `O`: OSType
- `U`: unicode16
- `u`: unsigned decimal
- `p`: pointer

Where to go from here?

Pat yourself on the back — this was another jam-packed round of what you can do with the `expression` command. Try exploring some of the other expression options yourself by executing `help expression` and see if you can figure out what they do.

Chapter 6: Thread, Frame & Stepping Around

You've learned how to create breakpoints, how to print and modify values, as well as how to execute code while paused in the debugger. But so far, you've been left high and dry on how to move around in the debugger and inspect data beyond the immediate. It's time to fix that!

In this chapter, you'll learn how to move the debugger in and out of functions while LLDB is currently paused.

This is a critical skill to have since you often want to inspect values as they change over time when entering or exiting snippets of code.

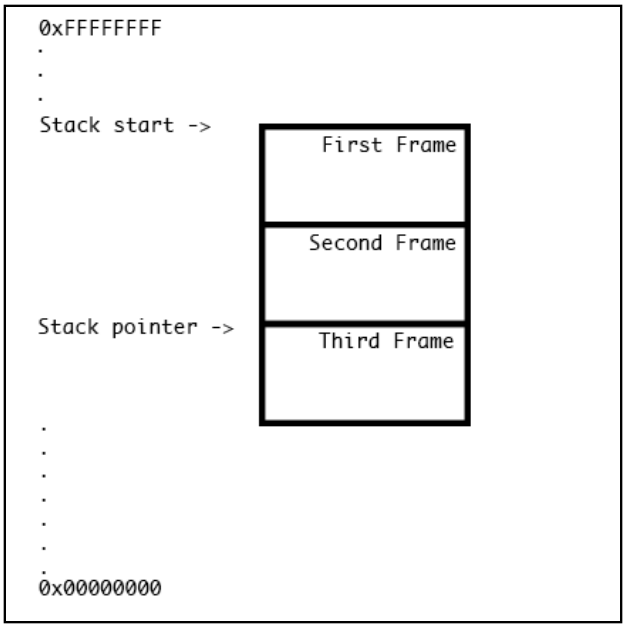
Stack 101

When a computer program executes, it stores values in the **stack** and the **heap**. Both have their merits. As an advanced debugger, you'll need to have a good understanding of how these work. Right now, let's take a brief look at the stack.

You may already know the whole spiel about what a stack is in computer science terms. In any case, it's worth having a basic understanding (or refresher) of how a process keeps track of code and variables when executing. This knowledge will come in handy as you're using LLDB to navigate around code.

The stack is a LIFO (Last-In-First-Out) queue that stores references to your currently executing code. This LIFO ordering means that whatever is added most recently, is removed first. Think of a stack of plates. Add a plate to the top, and it will be the one you take off first.

The **stack pointer** points to the current top of the stack. In the plate analogy, the stack pointer points to that top plate, telling you where to take the next plate from, or where to put the next plate on.



In this diagram, the high address is shown at the top (0xFFFFFFFF) and the low address is shown at the bottom (0x00000000) showcasing the stack would grow downwards.

Some illustrations like to have the high address at the bottom to match with the plate analogy as the stack would be shown growing upwards. However, I believe any diagrams showcasing the stack should be shown growing downwards from a high address because this will cause less headaches later on when talking about offsets from the stack pointer.

You'll take an in depth look at the stack pointer and other registers in Chapter 13, "Assembly and the Stack", but in this chapter you'll explore various ways to step through code that is on the stack.

Examining the stack's frames

You'll continue to use the **Signals** project for this chapter.

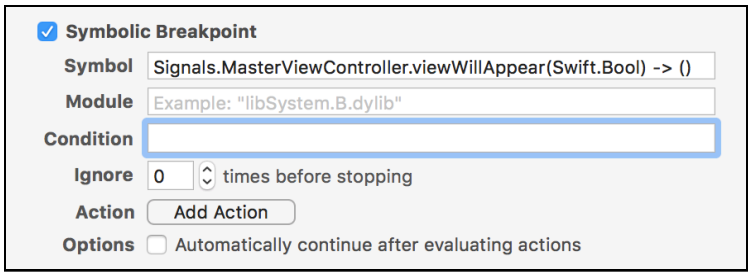
You'll glimpse some assembly in this chapter. Don't get scared! It's not that bad. However, be sure to use the iPhone X Simulator for this chapter since the assembly will be different if you were to generate the code on say, an actual iOS device.

This is because a device uses the ARM architecture, whereas the simulator uses your Mac's native instruction set, x86_64 (or i386 if you are compiling on something lower than the iPhone 5s Simulator).

Open the Signals project in Xcode. Next, add a symbolic breakpoint with the following function name. Be sure to honor the spaces in the function signature or else the breakpoint will not be recognized.

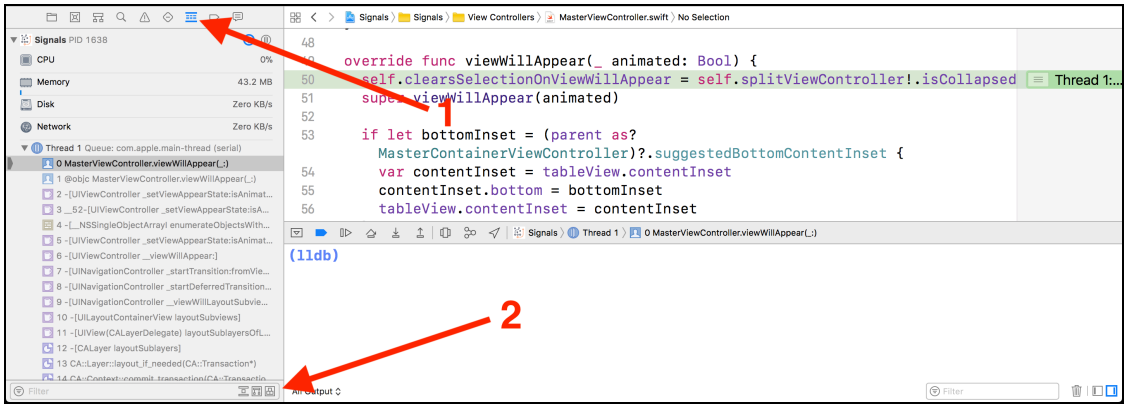
```
Signals.MasterViewController.viewWillAppear(Swift.Bool) -> ()
```

This creates a symbolic breakpoint on `MasterViewController`'s `viewWillAppear(_:)` method.



Build and run the program. As expected, the debugger will pause the program on the `viewWillAppear(_:)` method of `MasterViewController`. Next, take a look at the stack trace in the left panel of Xcode. If you don't see it already, click on the **Debug Navigator** in the left panel (alternatively, press **Command** + **7**, if you have the default Xcode keymap).

Make sure the three buttons in the bottom right corner are all disabled. These help filter stack functions to only functions you have source code for. Since you're learning about public as well as private code, you should always have these buttons disabled so you can see the full stack trace.



Within the Debug Navigator panel, the **stack trace** will appear, showing the list of **stack frames**, the first one being `viewWillAppear(_:)`. Following that is the Swift/Objective-C bridging method, `@objc MasterViewController.viewWillAppear(Bool) -> ()`:. This method is automatically generated so Objective-C can reach into Swift code.

After that, there's a few stack frames of Objective-C code coming from `UIKit`. Dig a little deeper, and you'll see some `C++` code belonging to `CoreAnimation`. Even deeper, you'll see a couple of methods all containing the name `CFRunLoop` that belong to `CoreFoundation`. Finally, to cap it all off, is the **`main`** function (yes, Swift programs still have a `main` function, it's just hidden from you).

The stack trace you see in Xcode is simply a pretty printed version of what LLDB can tell you. Let's see that now.

In the LLDB console, type the following:

```
(lldb) thread backtrace
```

You could also simply type `bt` if you wished, which does the same. It's actually a different command and you can see the difference if you pull out your trusty friend, `help`.

After the command above, you'll see a stack trace much like you see in Xcode's Debug Navigator.

Type the following into LLDB:

```
(lldb) frame info
```

You'll get a bit of output similar to the following:

```
frame #0: 0x000000010ba1f8dc
Signals`MasterViewController.viewWillAppear(animated=false,
self=0x00007fd286c0af10) at MasterViewController.swift:50
```

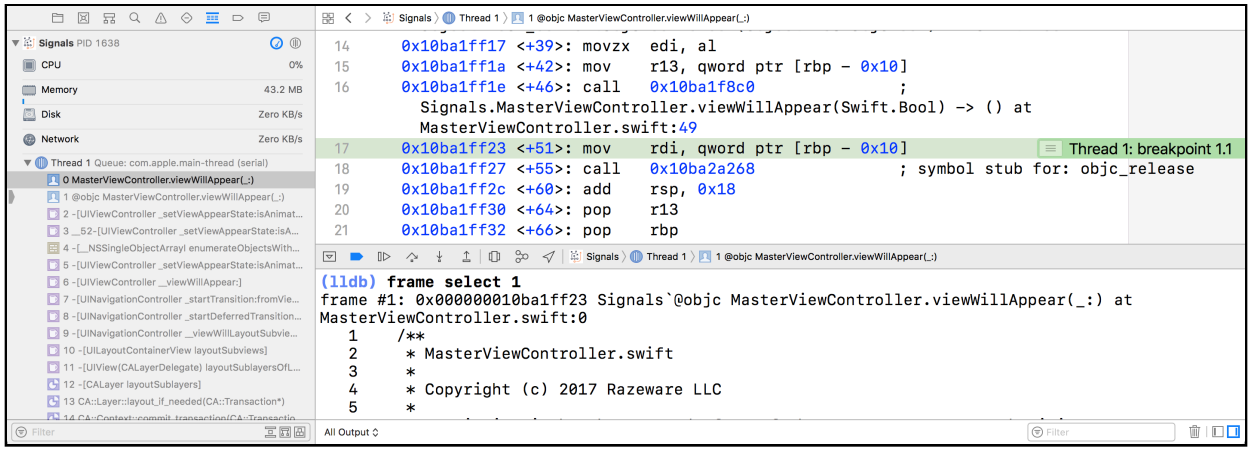
As you can see, this output matches the content found in the Debug Navigator. So why is this even important if you can just see everything from the Debug Navigator? Well, using the LLDB console gives you finer-grained control of what information you want to see. In addition, you'll be making custom LLDB scripts in which these commands will become very useful. It's also nice to know where Xcode gets its information from, right?

Taking a look back at the Debug Navigator, you'll see some numbers starting from 0 and incrementing as you go down the **call stack**. This numbering helps you associate which **stack frame** you're looking at. Select a different stack by typing the following:

```
(lldb) frame select 1
```

Xcode will jump to the `@objc` bridging method, the method located at index 1 in the stack. What's an `@objc` bridging method? It's a method that's generated by the Swift compiler to interact with Objective-C's dynamic nature. In earlier versions of Swift (Swift <= 3.2) any `NSObject` implied `@objc` bridging methods being generated. With the default build settings in Swift 4, even an Objective-C NSObject needs to have `@objc` (or `@objcMembers`) attribute for the Swift compiler to generate the bridging methods.

Provided you're using the Simulator and not an actual device, you'll get some assembly looking similar to the following.



Take note of the green line in the assembly. Right before that line is the **`callq`** instruction that is responsible for executing `viewWillAppear(_:)` you set a breakpoint on earlier.

Don't let the assembly blur your eyes too much. You're not out of the assembly woods just yet...

Stepping

When mastering LLDB, the three most important navigation actions you can do while the program is paused revolve around **stepping** through a program. Through LLDB, you can **step over**, **step in**, or **step out** of code.

Each of these allow you to continue executing your program's code, but in small chunks to allow you to examine how the program is executing.

Stepping over

Stepping over allows you to step to the next code statement (usually, the next line) in the context where the debugger is currently paused. This means if the current statement is calling another function, LLDB will run until this function has completed and returned.

Let's see this in action.

Type the following in the LLDB console:

```
(lldb) run
```

This will relaunch the Signals program without Xcode having to recompile. Neat! Xcode will stop on your symbolic breakpoint as before.

Next, type the following:

```
(lldb) next
```

The debugger will move one line forward. This is how you step over. Simple, but useful!

Stepping in

Stepping in means if the next statement is a function call, the debugger will move into the start of that function and then pause again.

Let's see this in action.

Relaunch the Breakpoints program from LLDB:

```
(lldb) run
```

Next, type the following:

```
(lldb) step
```

No luck. The program should've stepped in, because the line it's on contains a function call (well, actually it contains a few!).

In this case, LLDB acted more like a "step over" instead of a "step into". This is because LLDB will, by default, ignore stepping into a function if there are no debug symbols for that function. In this case, the function calls are all going into UIKit, for which you don't have debug symbols.

There is, however, a setting that specifies how LLDB should behave when stepping into a function for which no debug symbols exist. Execute the following command in LLDB to see where this setting is held:

```
(lldb) settings show target.process.thread.step-in-avoid-nodebug
```

If true, then stepping in will act as a step over in these instances. You can either change this setting (which you'll do in the future), or tell the debugger to ignore the setting, which you'll do now.

Type the following into LLDB:

```
(lldb) step -a0
```

This tells LLDB to step in regardless of whether you have the required debug symbols or not.

Stepping out

Stepping out means a function will continue for its duration then stop when it has returned. From a stack viewpoint, execution continues until the stack frame is popped off.

Run the Signals project again, and this time when the debugger pauses, take a quick look at the stack trace. Next, type the following into LLDB:

```
(lldb) finish
```

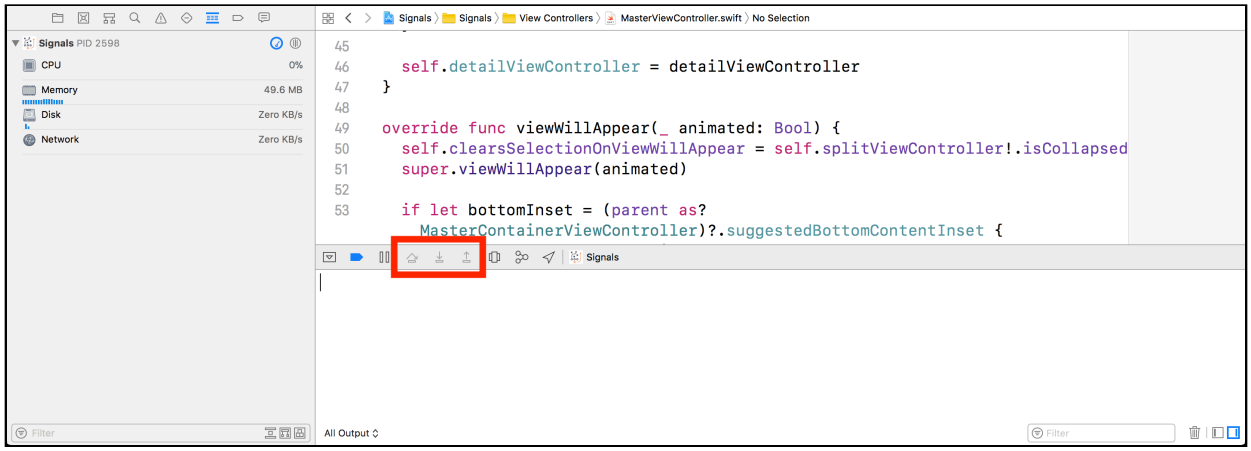
You'll notice that the debugger is now paused one function up in the stack trace. Try executing this command a few more times.

Remember, by simply pressing **Enter**, LLDB will execute the last command you typed. The `finish` command will instruct LLDB to step out of the current function.

Pay attention to the stack frames in the left panel as they disappear one by one.

Stepping in the Xcode GUI

Although you get much more finer-grained control using the console, Xcode already provides these options for you as buttons just above the LLDB console. These buttons appear when an application is running.



They appear, in order, as **step over**, **step in**, and **step out**.

Finally, the step over and step in buttons have one more cool trick. You can manually control the execution of different threads, by holding down **Control** and **Shift** while clicking on these buttons.

This will result in stepping through the thread on which the debugger is paused, while the rest of the threads remain paused. This is a great trick to have in the back of your toolbox if you are working with some hard-to-debug concurrency code like networking or something with Grand Central Dispatch.

Of course LLDB has the command line equivalent to do the same from the console by using the `--run-mode` option, or more simply `-m` followed by the appropriate option.

Examining data in the stack

A very interesting option of the `frame` command is the **`frame variable`** subcommand. This command will take the debug symbol information found in the headers of your executable (or a dYSM if your app is stripped... more on that later) and dump information out for that particular stack frame. Thanks to the debug information, the `frame variable` command can easily tell you the scope of all the variables in your function as well as any global variables within your program using the appropriate options.

Run the Signals project again and make sure you hit the `viewWillAppear(_:)` breakpoint. Next, navigate to the top of the stack by either clicking on the top stack frame in Xcode's Debug Navigator or by entering **`frame select 0`** in the console, or use LLDB's shorthand command **f 0**.

Next, type the following:

```
(lldb) frame variable
```

You'll get output similar to the following:

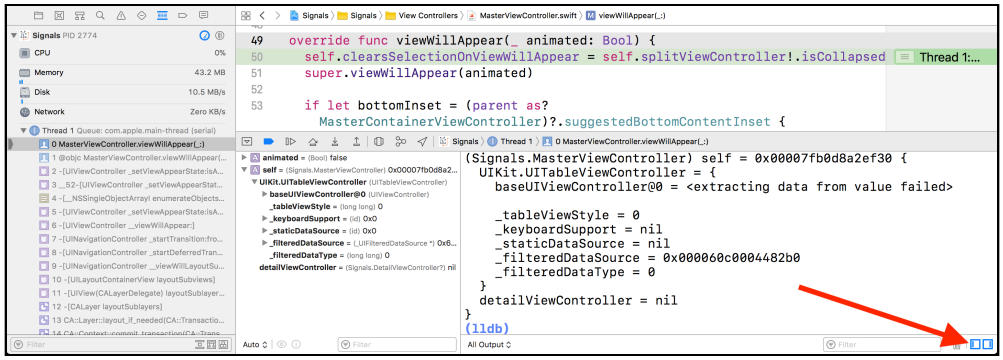
```
(Bool) animated = false
(Signals.MasterViewController) self = 0x00007fb3d160aad0 {
  UIKit.UITableViewController = {
    baseUIViewController@0 = <extracting data from value failed>

    _tableViewStyle = 0
    _keyboardSupport = nil
    _staticDataSource = nil
    _filteredDataSource = 0x000061800005f0b0
    _filteredDataType = 0
  }
  detailViewController = nil
}
```

This dumps the variables available to the current stack frame and line of code. If possible, it'll also dump all the instance variables, both public and private, from the current available variables.

You, being the observant reader you are, might notice the output of `frame variable` also matches the content found in the **Variables View**, the panel to the left of the console window.

If it's not already, expand the Variables View by clicking on the left icon in the lower right corner of Xcode. You can compare the output of `frame variable` to the Variables View. You might notice `frame variable` will actually give you more information about the ivars of Apple's private API than the Variables View will.



Next, type the following:

```
(lldb) frame variable -F self
```

This is an easier way to look at all the private variables available to MasterViewController. It uses the `-F` option, which stands for "flat".

This will keep the indentation to 0 and only print out information about `self`, in **MasterViewController.swift**.

You'll get output similar to the truncated output below:

```
self = 0x00007fff5540eb40
self =
self =
self =
self = {}
self.detailViewController = 0x00007fc728816e00
self.detailViewController.some =
self.detailViewController.some =
self.detailViewController.some = {}
self.detailViewController.some.signal = 0x00007fc728509de0
```

As you can see, this is an attractive way to explore public variables when working with Apple's frameworks.

Where to go from here?

In this chapter, you've explored stack frames and the content in them. You've also learned how to navigate the stack by stepping in, out, and over code.

There are a lot of options in the `thread` command you didn't cover. Try exploring some of them with the `help thread` command, and seeing if you can learn some cool options.

Take a look at the `thread until`, `thread jump`, and `thread return` subcommands. You'll use them later, but they are fun commands so give them a shot now to see what they do!

Chapter 7: Image

By now, you have a solid foundation in debugging. You can find and attach to processes of interest, efficiently create regular expression breakpoints to cover a wide range of culprits, navigate the stack frame and tweak variables using the `expression` command.

However, it's time to explore one of the best tools for finding code of interest through the powers of LLDB. In this chapter, you'll take a deep dive into the **`image`** command.

The `image` command is an alias for the **`target modules`** subcommand. The `image` command specializes in querying information about **modules**; that is, the code loaded and executed in a process. Modules can comprise many things, including the main executable, frameworks, or plugins. However, the majority of these modules typically come in the form of **dynamic libraries**. Examples of dynamic libraries include UIKit for iOS or AppKit for macOS.

The `image` command is great for querying information about any private frameworks and its classes or methods not publicly disclosed in these header files.

Wait... modules?

You'll continue using the `Signals` project. Fire up the project, build on the iPhone X Simulator and run.

Pause the debugger and type the following into the LLDB console:

```
(lldb) image list
```

This command will list all the modules currently loaded. You'll see a lot! The start of the list should look something like the following:

```
[  0] 1E1B0254-4F55-3985-92E4-B2B6916AD424 0x000000010e7e7000 /Users/
derekselander/Library/Developer/Xcode/DerivedData/Signals-
atjgadijglwyppbagqpvyvftavcw/Build/Products/Debug-iphonesimulator/
Signals.app/Signals
[  1] 002B0442-3D59-3159-BA10-1C0A77859C6A 0x000000011e7c8000 /usr/lib/
dyld
[  2] E991FA37-F8F9-39BB-B278-3ACF4712A994 0x000000010e817000 /
Applications/Xcode.app/Contents/Developer/Platforms/iPhoneOS.platform/
Developer/Library/CoreSimulator/Profiles/Runtimes/iOS.simruntime/
Contents/Resources/RuntimeRoot/usr/lib/dyld_sim
```

The first module is the app's main binary, `Signals`. The second and third modules pertain to the dynamic link editors (dyld). These to modules allow your program to load dynamic libraries into memory as well as the main executable in your process.

But there's a lot more in this list! You can filter out just those of interest to you. Type the following into LLDB:

```
(lldb) image list Foundation
```

The output will look similar to the following:

```
[  0] D153C8B2-743C-36E2-84CD-C476A5D33C72 0x000000010eb0c000 /
Applications/Xcode.app/Contents/Developer/Platforms/iPhoneOS.platform/
Developer/Library/CoreSimulator/Profiles/Runtimes/iOS.simruntime/
Contents/Resources/RuntimeRoot/System/Library/Frameworks/
Foundation.framework/Foundation
```

This is a useful way to find out information about just the module or modules you want.

Let's explore this output. There's a few interesting bits in there:

1. The module's UUID is printed out first (`D153C8B2-743C-36E2-84CD-C476A5D33C72`). The UUID is important for hunting down symbolic information and uniquely identifies the version of the Foundation module.

2. Following the UUID is the load address (`0x000000010eb0c000`). This identifies where the Foundation module is loaded into the `Signals` executable's process space.

3. Finally, you have the full path to where the module is located on disk.

Let's take a deeper dive into another common module, UIKit. Type the following into LLDB:

```
(lldb) image dump symtab UIKitCore -s address
```

This will dump all the symbol table information available for UIKitCore. It's more output than you can shake a stick at! This command sorts the output by the address in which the functions are implemented in the private UIKitCore module thanks to the `–s address` argument.

There's a lot of useful information in there, but you can't go reading all that, now can you? You need a way to effectively query the UIKitCore module with a flexible way to search for code of interest.

The **`image lookup`** command is perfect for filtering out all the data. Type the following into LLDB:

```
(lldb) image lookup –n "–[UIViewController viewDidLoad]"
```

This will dump out information relating just to UIViewController's `viewDidLoad` instance method. You'll see the name of the symbol relating to this method, and also where the code for that method is implemented inside the UIKitCore framework. This is good and all, but typing this is a little tedious and this can only dump out very specific instances.

This is where regular expressions come into play. The `–r` option will let you do a regular expression query. Type the following into LLDB:

```
(lldb) image lookup –rn UIViewController
```

Not only will this dump out all UIViewController methods, it'll also spit out results like `UIViewControllerBuiltinTransitionViewAnimator` since it contains the name `UIViewController`. You can be smart with the regular expression query to only spit out UIViewController methods. Type the following into LLDB:

```
(lldb) image lookup –rn '\[UIViewController\ '
```

Alternatively, you can use the `\s` meta character to indicate a space so you don't have to escape an actual space and surround it in quotes. The following expression is equivalent:

```
(lldb) image lookup –rn \[UIViewController\s
```

This is good, but what about categories? They come in the form of `UIViewController(CategoryName)`. Search for all `UIViewController` categories.

```
(lldb) image lookup –rn '\[UIViewController\(\w+\)\ '
```

.This is starting to get complicated. The backslash at the beginning says you want the literal character for "[", then UIViewController. Finally the literal character of "(" then one or more alphanumeric or underscore characters (denoted by `\w+`), then ")", followed by a space.

Working knowledge of regular expressions will help you to creatively query any public or private code in any of the modules loaded into your binary.

Not only does this print out both public and private code, this will also give you hints to the methods the `UIViewController` class overrides from its parent classes.

Hunting for code

Regardless of whether you're hunting for public or private code, sometimes it's just interesting trying to figure out how the compiler created the function name for a particular method. You briefly used the `image lookup` command above to find UIViewController methods. You also used it to hunt for how Swift property setters and getters are named in Chapter 4, "Stopping in Code."

However, there are many more cases where knowing how code is generated will give you a better understanding of where and how to create breakpoints for code you're interested in. One particularly interesting example to explore is the method signature for Objective-C's blocks.

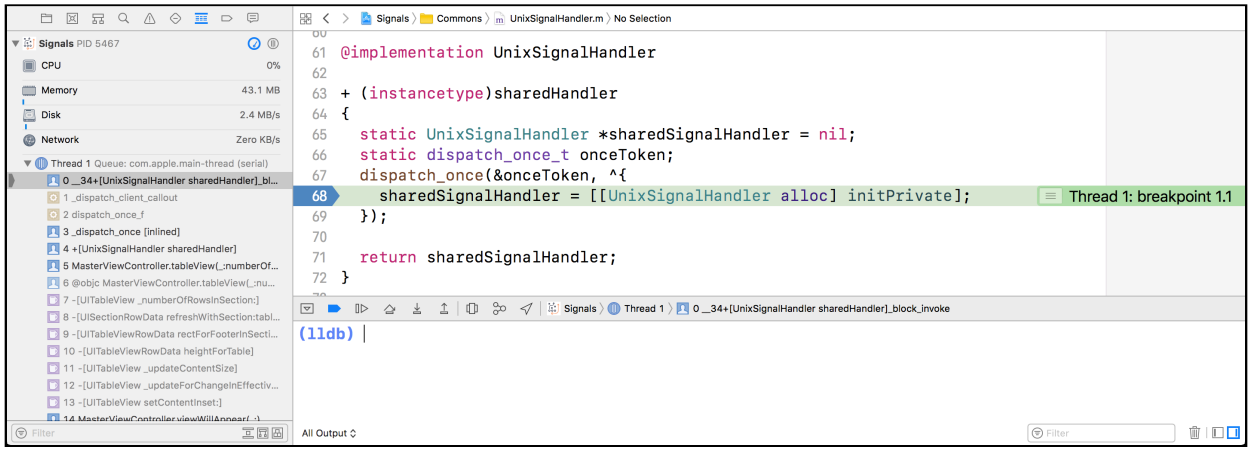
So what's the best way to search for a method signature for an Objective-C block? Since you don't have any clue on where to start searching for how blocks are named, a good way to start is by putting a breakpoint inside a block and then inspecting from there.

Open **UnixSignalHandler.m**, then find the singleton method **`sharedHandler`**. Within the function, look for the following code:

```
dispatch_once(&onceToken, ^{
  sharedSignalHandler = [[UnixSignalHandler alloc] initPrivate];
});
```

Put a breakpoint using the Xcode GUI in the line beginning with **`sharedSignalHandler`**.

Then build and run. Xcode will now pause on the line of code you just set a breakpoint on. Check out the top stack frame in the debugging window.



You can find the name of the function you're in using Xcode's GUI. In the Debug Navigator you'll see your stack trace and you can look at frame 0. That's a little hard to copy and paste (well, impossible, actually). Instead, type the following into LLDB:

```
(lldb) frame info
```

You'll get output similar to the following:

```
frame #0: 0x000000010f9b45a0 Commons`__34+[UnixSignalHandler
sharedHandler]_block_invoke(.block_descriptor=0x000000010f9ba200) at
UnixSignalHandler.m:72
```

As you can see, the full function name is **`__34+[UnixSignalHandler sharedHandler]_block_invoke`**.

There's an interesting portion to the function name, **`_block_invoke`**. This might be the pattern you need to help uniquely identify blocks in Objective-C. Type the following into LLDB:

```
(lldb) image lookup -rn _block_invoke
```

This will do a regular expression search for the word **`_block_invoke`**. It will treat everything before and after the phrase as a wildcard.

But wait! You accidentally printed out *all* the Objective-C blocks loaded into the program. This search included anything from UIKit, Foundation, iPhoneSimulator SDK, etc. You should limit your search to only search for the Signals module.

Type the following into LLDB:

```
(lldb) image lookup -rn _block_invoke Signals
```

Nothing is printed out. What gives? Open the right Xcode panel and click on the **File Inspector**. Alternatively, press ⌘ + **Option** + **1** if you have the default Xcode keymap.

If you look to where **UnixSignalHandler.m** is compiled, you'll see it's actually compiled into the **Commons** framework. So, redo that search and look for Objective-C blocks in the Commons module. Type the following into LLDB:

```
(lldb) image lookup -rn _block_invoke Commons
```

Finally, you'll get some output!

You'll now see all the Objective-C blocks that you've searched for in the Commons framework.

Now, let's create a breakpoint to stop on a subset of these blocks you've found. Type the following into LLDB:

```
(lldb) rb appendSignal.*_block_invoke -s Commons
```

> **Note**: There is a subtle difference between searching for code in a module versus breaking in code for a module. Take the above commands as an example. When you wanted to search for all blocks in the Commons framework, you used `image lookup -rn _block_invoke Commons`. When you wanted to make breakpoints for blocks in the Commons framework, you used `rb appendSignal.*block_invoke -s Commons`. Take note of the `-s` argument vs the space.

The idea is this breakpoint will hit on any block within the `appendSignal` method.

Resume the program by clicking the play button or typing `continue` into LLDB. Jump over to Terminal and type the following:

```
pkill -SIGIO Signals
```

The signal you sent the program will be processed. However, before the signal gets visually updated to the tableview, your regex breakpoint will get hit.

The first breakpoint you will hit will be in:

```
__38-[UnixSignalHandler appendSignal:sig:]_block_invoke
```

Go past this by continuing the debugger.

Next you'll hit a breakpoint in:

```
__38-[UnixSignalHandler appendSignal:sig:]_block_invoke_2
```

There's an interesting item to note about this function name compared to the first; notice the number **2** in the method name. The compiler uses a base of `<FUNCTION_NAME>_block_invoke` for blocks defined within the function called `<FUNCTION_NAME>`. However, when there's more than one block in the function, a number is appended to the end to denote this.

As you learned in the previous chapter, the `frame variable` command will print all known local variable instances to a particular function. Execute that command now to see the reference found in this particular block. Type the following into LLDB:

```
(lldb) frame variable
```

The output will look similar to the following:

```
(__block_literal_5 *)  = 0x0000608000275e80
(int) sig = <read memory from 0x41 failed (0 of 4 bytes read)>

(siginfo_t *) siginfo = <read memory from 0x39 failed (0 of 8 bytes
read)>

(UnixSignalHandler *const) self = <read memory from 0x31 failed (0 of 8
bytes read)>
```

Those read memory failures don't look good! Step over once, either using the Xcode GUI or by typing `next` in LLDB. Next, execute `frame variable` again in LLDB. This time you'll see something similar to the following:

```
(__block_literal_5 *)  = 0x0000608000275e80
(int) sig = 23
(siginfo_t *) siginfo = 0x00007fff587525e8
(UnixSignalHandler *) self = 0x000061800007d440
(UnixSignal *) unixSignal = 0x000000010bd9eebe
```

You needed to step over one statement, so the block executed some initial logic to setup the function, also known as the function prologue. The function prologue is a topic related to assembly, which you'll learn about in Section II.

This is actually quite interesting. First you see an object which references the block that's being invoked. In this case it's the type `__block_literal_5`. Then there are the `sig` and `siginfo` parameters that were passed into the Objective-C method where this block is invoked from. How did these get passed into the block?

Well, when a block is created, the compiler is smart enough to figure out what parameters are being used by it. It then creates a function that takes these as parameters. When the block is invoked, it's this function that is called, with the relevant parameters passed in.

Type the following into LLDB:

```
(lldb) image lookup -t  __block_literal_5
```

You'll get something similar to the following:

```
Best match found in /Users/derekselander/Library/Developer/Xcode/
DerivedData/Signals-efqxsbqzgzcqqvhjgzgeabtwfufy/Build/Products/Debug-
iphonesimulator/Signals.app/Frameworks/Commons.framework/Commons:
id = {0x100000cba}, name = "__block_literal_5", byte-size = 52, decl =
UnixSignalHandler.m:123, compiler_type = "struct __block_literal_5 {
    void *__isa;
    int __flags;
    int __reserved;
    void (*__FuncPtr)();
    __block_descriptor_withcopydispose *__descriptor;
    UnixSignalHandler *const self;
    siginfo_t *siginfo;
    int sig;
}"
```

This is the object that defines the block! Neat!

As you can see, this is almost as good as a header file for telling you how to navigate the memory in the block. Provided you cast the reference in memory to the type `__block_literal_5`, you can easily print out all the variables referenced by the block.

Start by getting the stack frame's variable information again by typing the following:

```
(lldb) frame variable
```

Next, find the address of the `__block_literal_5` object and print it out like so:

```
(lldb) po ((__block_literal_5 *)0x0000618000070200)
```

You should see something similar to the following:

```
<__NSMallocBlock__: 0x0000618000070200>
```

If you don't, make sure the address you're casting to a `__block_literal_5` is the address of your block as it will differ each time the project is run.

> **Note**: Bug alert in lldb-900.0.57 where LLDB will incorrectly dereference the `__block_literal_5` pointer when executing the `frame variable` command. This means that the pointer output of `(__block_literal_5 *)` will give the *class* **NSMallocBlock** instead of the instance of **NSMallocBlock**. If you are getting the class description instead of an instance description, you can get around this by either referencing the `RDI` register immediately at the start of the function, or obtain the instance of the `__NSMallocBlock__` via `x/gx '$rbp - 32'` if you are further into the function.

Now you can query the members of the `__block_literal_5` struct. Type the following into LLDB:

```
(lldb) p/x ((__block_literal_5 *)0x0000618000070200)->__FuncPtr
```

This will dump the location of the function pointer for the block. The output will look like the following:

```
(void (*)()) $1 = 0x000000010756d8a0 (Commons`__38-[UnixSignalHandler
appendSignal:sig:]_block_invoke_2 at UnixSignalHandler.m:123)
```

The function pointer for the block points to the function which is run when the block is invoked. It's the same address that is being executed right now! You can confirm this by typing the following, replacing the address with the address of your function pointer printed in the command you last executed:

```
(lldb) image lookup -a 0x000000010756d8a0
```

This uses the `-a` (address) option of `image lookup` to find out which symbol a given address relates to.

Jumping back to the block struct's members, you can also print out all the parameters passed to the block as well. Type the following, again replacing the address with the address of your block:

```
(lldb) po ((__block_literal_5 *)0x0000618000070200)->sig
```

This will output the signal number that was sent in as a parameter to the block's parent function.

There is also a reference to the `UnixSignalHandler` in a member of the struct called `self`. Why is that? Take a look at the block and hunt for this line of code:

```
[(NSMutableArray *)self.signals addObject:unixSignal];
```

It's the reference to `self` the block captured, and uses to find the offset of where the `signals` array is. So the block needs to know what `self` is. Pretty cool, eh?

By the way, you can dump out the full struct with the `p` command and dereferencing the pointer like so:

```
(lldb) p *(__block_literal_5 *)0x0000618000070200
```

Using the `image dump symfile` command in combination with the module is a great way to learn how a certain unknown data type works. It's also a great tool to understand how the compiler generates code for your sources.

Additionally, you can inspect how blocks hold references to pointers outside the block — a very useful tool when debugging memory retain cycle problems.

Snooping around

OK, you've discovered how to inspect a private class's instance variables in a static manner, but that block memory address is too tantalizing to be left alone. Try printing it out and exploring it using dynamic analysis. Type the following, replacing the address with the address of your block:

```
po 0x0000618000070200
```

LLDB will dump out a class indicating it's an Objective-C class.

```
<__NSMallocBlock__: 0x618000070200>
```

This is interesting. The class is `__NSMallocBlock__`. Now that you've learned how to dump methods for both private and public classes, it's time to explore what methods `__NSMallocBlock__` implements. In LLDB, type:

```
(lldb) image lookup -rn __NSMallocBlock__
```

Nothing. Hmm. This means `__NSMallocBlock__` doesn't override any methods implemented by its super class. Type the following in LLDB to figure out the parent class of `__NSMallocBlock__`.

```
(lldb) po [__NSMallocBlock__ superclass]
```

This will produce a similarly named class named **`__NSMallocBlock`** — notice the lack of trailing underscores. What can you find out about this class? Does this class implement or override any methods? Type the following into LLDB:

```
(lldb) image lookup -rn __NSMallocBlock
```

The methods dumped by this command seems to indicate that `__NSMallocBlock` is responsible for memory management, since it implements methods like `retain` and `release`. What is the parent class of `__NSMallocBlock`? Type the following into LLDB:

```
(lldb) po [__NSMallocBlock superclass]
```

You'll get another class named `NSBlock`. What about this class? Does it implement any methods? Type the following into LLDB:

```
(lldb) image lookup -rn 'NSBlock\ '
```

Notice the backslash and space at the end. This ensures there are no other classes that will match this query — remember, without it, a different class could be returned that *contains* the name `NSBlock`. A few more methods will be spat out. One of them, **`invoke`**, looks incredibly interesting:

```
Address: CoreFoundation[0x000000000018fd80] (CoreFoundation.__TEXT.__text
+ 1629760)
        Summary: CoreFoundation`-[NSBlock invoke]
```

You're now going to try to invoke this method on the block. However, you don't want the block to disappear when the references that are retaining this block release their control, thus lowering the `retainCount`, and potentially deallocating the block.

There's a simple way to hold onto this block — just **retain** it! Type the following into LLDB, replacing the address with the address of your block:

```
(lldb) po id $block = (id)0x0000618000070200
(lldb) po [$block retain]
(lldb) po [$block invoke]
```

For the final line, you'll see the following output:

```
Appending new signal: SIGIO
 nil
```

This shows you the block has been invoked again! Pretty neat!

It only worked because everything was already set up in the right way for the block to be invoked, since you're currently paused right at the start of the block.

This type of methodology for exploring both public and private classes, and then exploring what methods they implement, is a great way to learn what goes on underneath the covers of a program. You'll later use the same process of discovery for methods and then analyze the assembly these methods execute, giving you a very close approximation of the source code of the original method.

Private debugging methods

The `image lookup` command does a beautiful job of searching for private methods as well the public methods you've seen throughout your Apple development career. However, there are some hidden methods which are quite useful when debugging your own code.

For example, a method beginning with _ usually denotes itself as being a private (and potentially important!) method.

Let's try to search for any Objective-C methods in all of the modules that begin with the underscore character and contain the word "description" in it.

Build and run the project again. When your breakpoint in `sharedHandler` is hit, type the following into LLDB:

```
(lldb) image lookup -rn (?i)\ _\w+description\]
```

This regular expression is a bit complex so let's break it down.

The expression searches for a space (`\`) followed by an underscore (_). Next, the expression searches for one or more alphanumeric or underscore characters (`\w+`) followed by the word `description`, followed by the `]` character.

The beginning of the regular expression has an interesting set of characters, `(?i)`. This states you want this to be a case insensitive search.

This regular expression has backslashes prepending characters. This means you want the literal character, instead of its regular expression meaning. It's called "escaping". For example, in a regular expression, the `]` character has meaning, so to match the literal "]" character, you need to use `\]`.

The exception to this in the regular expression above is the `\w` character. This is a special search item returning an alphanumeric character or an underscore (i.e. _, a-z, A-Z, 0-9).

If you had the deer in the headlights expression when reading this line of code, it's strongly recommended to carefully scan https://docs.python.org/2/library/re.html to brush up on your regular expression queries; it's only going to get more complicated from here on out.

Carefully scan through the output of `image lookup`. It's often this tedious scanning that gives you the best answers, so please make sure you go through all the output.

You'll notice a slew of interesting methods belonging to an `NSObject` category named **`IvarDescription`** belonging in UIKit.

Redo the search so only contents in this category get printed out. Type the following into LLDB:

```
(lldb) image lookup -rn NSObject\(IvarDescription\)
```

The console will dump out all the methods this category implements. Of the group of methods, there are a couple very interesting methods that stand out:

```
_ivarDescription
_propertyDescription
_methodDescription
_shortMethodDescription
```

Since this category is on `NSObject`, any subclass of `NSObject` can use these methods. This is pretty much everything, of course!

Execute the `_ivarDescription` on the `UIApplication` Objective-C class. Type the following into LLDB:

```
(lldb) po [[UIApplication sharedApplication] _ivarDescription]
```

You'll get a slew of output since `UIApplication` holds many instance variables behind the scenes. Scan carefully and find something that interests you. Don't come back to reading this until you find something of interest. This is important.

After carefully scanning the output, you can see a reference to the private class `UIStatusBar`. Which Objective-C setter methods does `UIStatusBar` have, I hear you ask? Let's find out! Type the following into LLDB:

```
(lldb) image lookup -rn '\[UIStatusBar\ set'
```

This dumps all the setter methods available to `UIStatusBar`. In addition to the declared and overriden methods available in `UIStatusBar`, you have access to all the methods available to its parent class. Check to see if the `UIStatusBar` is a subclass of the `UIView` class

```
(lldb) po (BOOL)[[UIStatusBar class] isSubclassOfClass:[UIView class]]
```

Alternatively, you can repeatedly use the `superclass` method to jump up the class hierarchy. As you can see, it looks like this class is a subclass of UIView, so the `backgroundColor` property is available to you in this class. Let's play with it.

First, type the following into LLDB:

```
(lldb) po [[UIApplication sharedApplication] statusBar]
```

You'll see something similar to the following:

```
<UIStatusBar_Modern: 0x7fdcf3c0f090; frame = (0 0; 375 44); autoresize =
W+BM; layer = <CALayer: 0x60c000036640>>
```

This prints out the `UIStatusBar` instance for your app. Next, using the address of the status bar, type the following into LLDB:

```
(lldb) po [0x7fdcf3c0f090 setBackgroundColor:[UIColor purpleColor]]
```

In LLDB, remove any of the previous breakpoints you created.

```
(lldb) breakpoint delete
```

Continue the app and see the beauty you've unleashed upon the world through your fingertips!



Not the prettiest of apps now, but at least you've managed to inspect a private method and used it to do something fun!

Where to go from here?

As a challenge, try figuring out a pattern using `image lookup` to find all Swift closures within the Signals module. Once you do that, create a breakpoint on every Swift closure within the Signals module. If that's too easy, try looking at code that can stop on **didSet**/**willSet** property helpers, or do/try/catch blocks.

Also, try looking for more private methods hidden away in Foundation and UIKit. Have fun!

Need another challenge?

Using the private UIKitCore NSObject category method `_shortMethodDescription` as well as your `image lookup -rn` command, search for the class that's responsible for displaying time in the upper left corner of the status bar and change it to something more amusing. Drill into subviews and see if you can find it using the tools given so far.



Chapter 8: Watchpoints

You've learned how to create breakpoints on executable code; that is, memory that has read and execute permissions. But using only breakpoints leaves out an important component to debugging — you can monitor when the instruction pointer executes an address, but you can't monitor when memory is being read or written to. You can't monitor value changes to instantiated Swift objects on the heap, nor can you monitor reads to a particular address (say, a hardcoded string) in memory. This is where a **watchpoint** comes into play.

A watchpoint is a special type of breakpoint that can monitor reads or writes to a particular value in memory and is not limited to executable code as are breakpoints. However, there are limitations to using watchpoints: there are a finite amount of watchpoints permitted per architecture (typically 4) and the "watched" size of memory usually caps out at 8 bytes.

Watchpoint best practices

Like all debugging techniques, a watchpoint is a type of tool in the debugging toolbox. You'll likely not use this tool very often, but it can be extremely useful in certain situations. Watchpoints are great for:

- Tracking an allocated Swift/Objective-C object when you don't know how a property is getting set, i.e. via direct ivar access, Objective-C property setter method, Swift property setter method, hardcoded offset access, or other methods.
- Monitoring when a hardcoded string is being utilized, such as in a `print`/`printf`/`NSLog`/`cout` function call.
- Monitor the instruction pointer for a particular type of assembly instruction.

Finding a property's offset

Watchpoints are great for discovering how a particular piece of memory is being written to. A pratical example of this is when a value is written to a previously allocated instance created from the heap, such as in an Objective-C/Swift class.

Fortunately, Swift automatically wraps all writes to offsets in a property's setter method; in Swift, *you don't have direct access to the ivar!* This means that watchpoints for Swift might not always be necessary since your best bet could be a breakpoint with the Swift setter syntax found in Chapter 4, "Stopping in Code". In fact, Swift also gives you the `didGet` and `didSet` methods which provide an attractive alternative to using watchpoints in Swift. The same does not get carried over to the C/ObjC/ObjC++ family, where a value in memory could be modified directly through the ivar, through the property setter, or through a hardcoded offset. This means you can't always rely on Objective-C's `set{PropertyName}:`'s breakpoint syntax to catch when a value is being set.

For this example, you'll see watchpoints in action by using one in lieu of a breakpoint to catch a particular write to memory.

Open up the **Signals** application in the starter directory for this chapter and run the program using the iOS 12 iPhone X Simulator.

Once running, pause the application and head over to the LLDB console. Type the following:

```
(lldb) language objc class-table dump UnixSignalHandler -v
```

This will dump the Objective-C class layout of `UnixSignalHandler`. The output will look similar to the following:

```
isa = 0x10e843d90 name = UnixSignalHandler instance size = 56 num ivars =
4 superclass = NSObject
  ivar name = source type = id size = 8 offset = 24
  ivar name = _shouldEnableSignalHandling type = bool size = 1 offset =
32
  ivar name = _signals type = id size = 8 offset = 40
  ivar name = _sharedUserDefaults type = id size = 8 offset = 48
  instance method name = setShouldEnableSignalHandling: type = v20@0:8B16
 ...
```

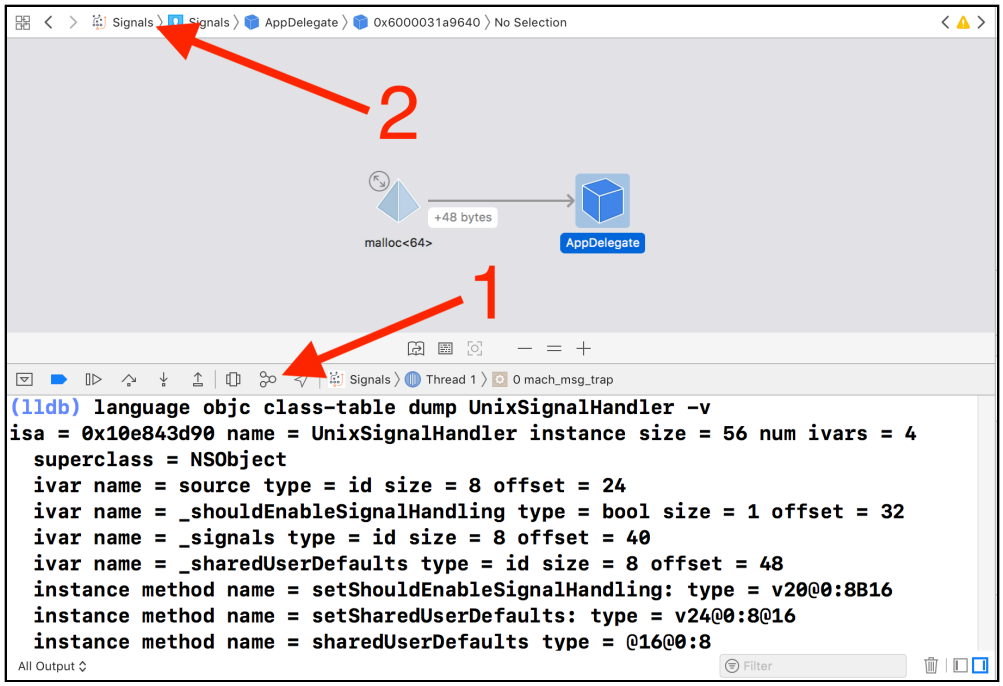
Check out **`_shouldEnableSignalHandling`**, whose offset is 32 bytes and whose size is 1 byte (yes, a byte, *NOT a bit*).

This means that if you know where an instance of the `UnixSignalHandler` class is located on the heap, you can add 32 bytes to that address to get the location where `_shouldEnableSignalHandling` is stored in an instance of `UnixSignalHandler`.

> **Note**: The LLDB command "`language objc class-table dump`" is a little buggy and won't work on Swift classes... even though a Swift class, on Apple platforms, inherits from an Objective-C class. If you aren't a fan of this broken command, you can check out the **`dclass`** command found here: https://github.com/DerekSelander/LLDB/blob/master/lldb_commands/dclass.py. The `dclass` command works on both Objective-C and Swift classes, and produces much cleaner output.

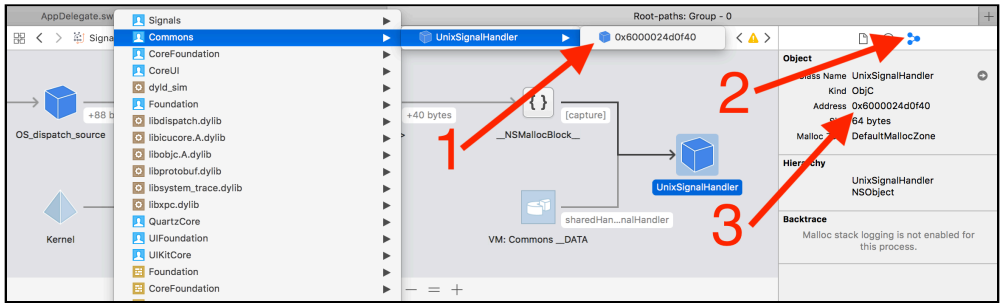
Now that you know the offset to find the `_shouldEnableSignalHandling` ivar on an instance, it's time to find the instance of the `UnixSignalHandler` singleton. In Xcode, tap on the **Debug Memory Graph** button located on the top of the debug console.

Once tapped, select the **Signals** project, then select the **Commons** framework (the framework responsible for implementing the `UnixSignalHandler`).



Drill down and you'll see the instance of the `UnixSignalHandler` both visually in Xcode and the memory address. Make sure to also open the inspectors on the right side and select the **Show the Memory Inspector** option.

Once you have the instance, copy the memory address of the `UnixSignalHandler` into your clipboard.



On my particular instance of the Signals program, I can see that the singleton instance of `UnixSignalHandler` has a heap address value starting at **0x6000024d0f40**, but note that yours will most likely be different.

> **Note**: Without dwelling too long on the fact that the author's debugging scripts may provide a better debugging experience than what Xcode can currently deliver, if you're not a fan of all the GUI clicking you just performed, please check out the **search** command here https://github.com/DerekSelander/LLDB/blob/master/lldb_commands/search.py. This command can enumerate the heap for specific Objective-C classes and is frankly, a more powerful and feature rich than Xcode's GUI equivalent.

Through LLDB, Add your instance value to 32 to find the location of the `_shouldEnableSignalHandling` ivar. Format the output in hexadecimal using LLDB's `p/x` (print hexadecimal) command.

```
(lldb) p/x 0x6000024d0f40 + 32
(long) $0 = 0x00006000024d0f60
```

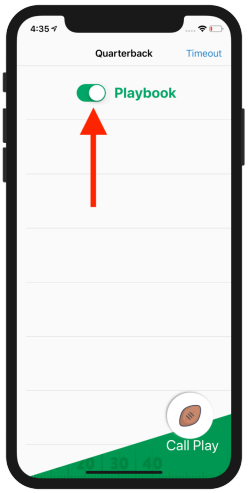
`0x00006000024d0f60` is the location of interest. Time to put a watchpoint on it!

In LLDB, type the following. Remember to replace your own calculated offset value of `UnixSignalHandler`:

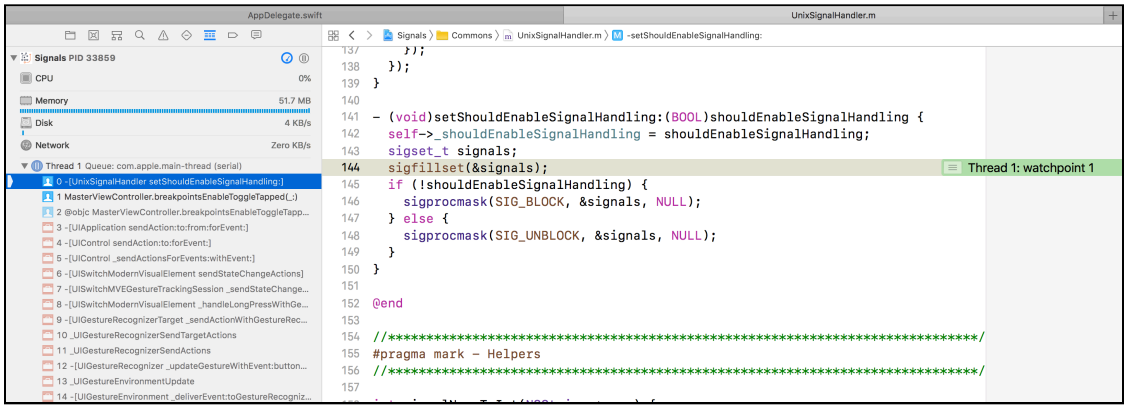
```
(lldb) watchpoint set expression -s 1 -w write -- 0x00006000024d0f60
```

This creates a new watchpoint that monitors address `0x00006000024d0f60`, whose size monitors a 1 byte range (thanks to the `-s 1` argument) and only stops if the value gets set (`-w write`). The **`-w`** argument can monitor `read`, `read_write` or `write` occurrences in memory.

Now that we have the appropriate plumbing in LLDB to monitor this change, it's time to trigger the event throug the Simulator. In the Signals project, tap on the playbook `UISwitch` button.



The Signals project will be suspended. Take a gander over to the left hand side of Xcode to view the stack trace and see how the program got stopped.



What caused the watchpoint

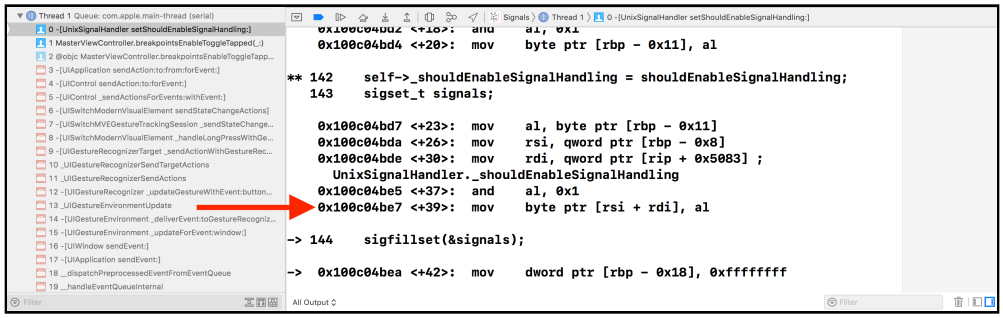
What exactly caused the watchpoint to be triggered? Caffeinate up, you'll be looking at a bit of assembly now. To find out, use LLDB to disassemble the current method.

```
(lldb) disassemble -F intel -m
```

This will print the current frame's disassembly in Intel format (more on this and assembly in Section II). In addition, you specified the **`-m`** option to show the assembly and source code as mixed. This will give you a better indication of how the assembly relates to the sourcecode.

Scan the output where the program counter is currently stopped at by the `->`. You'll see a `->` for both the assembly and the source outputs, but in reality, these are both the same.

It's the assembly instruction immediately above the program counter `->` line which is over interest to us.



In my case, I got the following instruction:

```
0x100c04be7 <+39>:  mov    byte ptr [rsi + rdi], al
```

> **Note**: A new version of Clang could change the assembly output. If that's the case, use this example as a guide to figure out your unique assembly instruction.

You don't need to know the specifics to x86_64 assembly yet (that's in Section II), but the expression is equivalent to the following:

```
*(BOOL *)(rsi + rdi) = al
```

You can prove that this will be the `UnixSignalHandler` instance `+ 32` offset by typing the following into LLDB:

```
(lldb) p/x $rsi + $rdi
```

This will produce the address of the watchpoint you created earlier. In fact, you can get that address of the previously created watchpoint by typing:

```
(lldb) watchpoint list
```

Still need convincing this is the instance of the `UnixSignalHandler`? Type the following to retrieve the original instance:

```
(lldb) po $rsi + $rdi - 32
<UnixSignalHandler: 0x6000024d0f40>
```

As you can see, the (0x6000024d0f20 + 32) memory address was modified by the **AL** register, which caused the watchpoint to trigger. This assembly instruction was the result of the following line in the sourcecode:

```
self->_shouldEnableSignalHandling = shouldEnableSignalHandling;
```

There was an overriden Objective-C property setter, which performed direct ivar access to the value. Although an Objective-C property setter breakpoint would have caught this in this particular example, you might not always be so lucky.

As you can see, the setup takes longer, but watchpoints can be much more powerful. This is why watchpoints are a great tool to use when your initial breakpoint strategies fail.

The Xcode GUI watchpoint equivalent

Xcode provides a GUI for setting watchpoints. You could perform the equivalent of the above methods by setting a breakpoint on the creation method of the `UnixSignalHandler` singleton, then set a watchpoint via the GUI. First though, you need to delete the previous watchpoint.

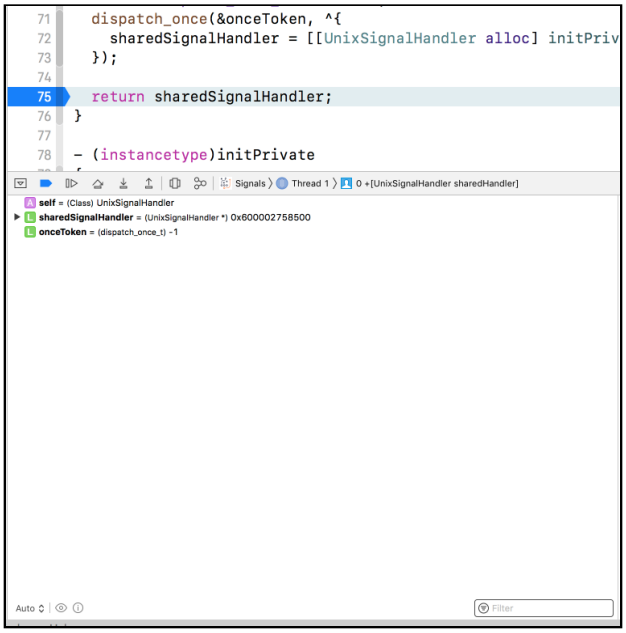
In LLDB, delete the watchpoint, then resume execution:

```
(lldb) watchpoint delete
About to delete all watchpoints, do you want to do that?: [Y/n] Y
All watchpoints removed. (1 watchpoints)
(lldb) c
Process 68247 resuming
```

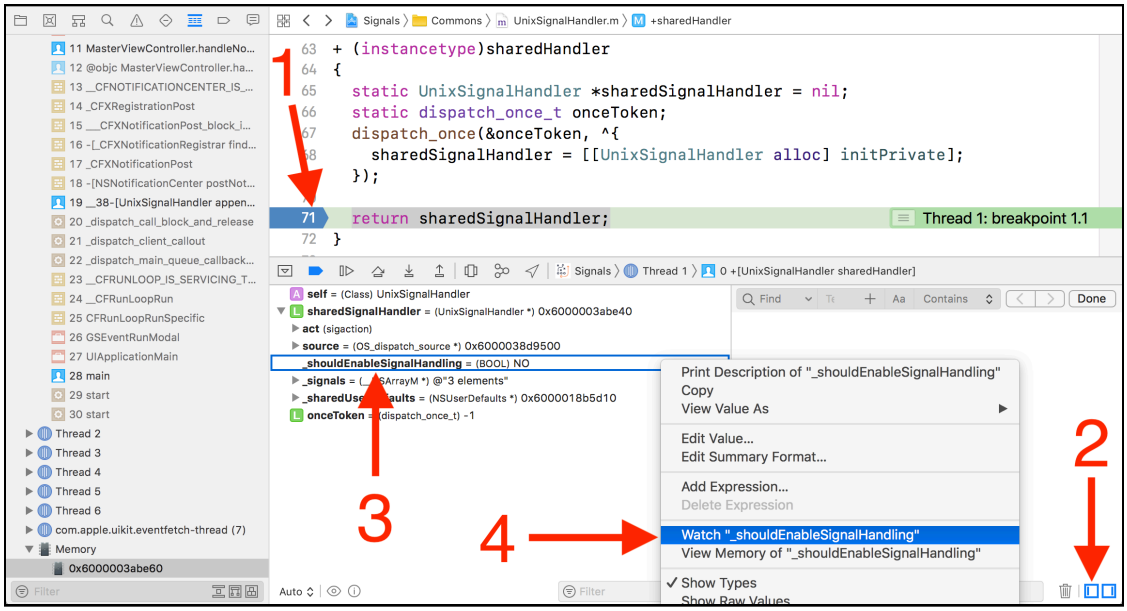
In the Signals program, make sure the Playbook `UISwitch` is flicked back to on. Once on, navigate to **UnixSignalHandler.m** and set a GUI breapoint at the end of the function that returns the singleton instance.

Control should suspend since you've added a breakpoint to a callback function that monitors breakpoints and references that code. If not, make sure your Playbook `UISwitch` is active.

Once control is suspended, make sure your **Variables View** is visible. It's found in the lower right corner of Xcode.



In the Variables View, drill down into the **sharedSignalHandler** instance, then right click on the **_shouldEnableSignalHandling** variable. Select **Watch _shouldEnableSignalHandling**.



Resume control of the program through Xcode or LLDB. Test out the newly created watchpoint by tapping the Playbook `UISwitch` yet again in the Simulator.

Other watchpoint tidbits

Fortunately, the syntax for watchpoints is very similar to the syntax for breakpoints. You can `delete`, `disable`, `enable`, `list`, `command`, or `modify` them just as you would using LLDB's breakpoint syntax.

The more interesting ones of the group are the `command` and `modify` actions. The `modify` command can add a condition to trigger the watchpoint only if it's `true`. The `command` action lets you perform a unique command whenever the watchpoint gets triggered.

For example, let's say you wanted to the previous watchpoint to only stop when the new value is set to 0.

First, find the watchpoint ID to modify:

```
(lldb) watchpoint list  -b
Number of supported hardware watchpoints: 4
Current watchpoints:
Watchpoint 2: addr = 0x60000274ee20 size = 1 state = enabled type = w
```

This says to list all the watchpoints in a "brief" (`-b`) format. You can see the Watchpoint ID is **2**. From there modify, Watchpoint ID 2:

```
(lldb) watchpoint modify 2 -c '*(BOOL*)0x60000274ee20 == 0'
```

This will modify Watchpoint ID 2 to only stop if the new value of `_shouldEnableSignalHandling` is set to `false`.

If you omit the Watchpoint ID in the above example (the 2), it will be applied to every valid watchpoint in the process.

One more example before you wrap this chapter up! Instead of conditionally stopping when `_shouldEnableSignalHandling` is set to 0, you can simply have LLDB print the stack trace everytime it's set.

Remove all watchpoint conditions like so:

```
(lldb) watchpoint modify 2
```

This will remove the condition you previously created. Now add a command to print the backtrace, then continue.

```
(lldb) watchpoint command add 2
Enter your debugger command(s).  Type 'DONE' to end.
> bt 5
> continue
> DONE
```

Instead of conditionally stopping, the watchpoint will print the first five stack frames in the LLDB console, then continue.

Once you get bored of seeing all that output, you can remove this command by typing:

```
(lldb) watchpoint command delete 2
```

And there you have it! Watchpoints in a nutshell.

Where to go from here?

Watchpoints tend to play very nicely with those who understand how an executable is laid out in memory. This layout, known as Mach-O, will be discussed in detail in Chapter 18, “Hello, Mach-O”. Combining this knowledge with watchpoints, you can watch when strings are referenced, or when static pointers are intialized, without having to tediously track the locations at runtime.

But for now, just remember that you have a great tool to use when you need to hunt for how something is created and your breakpoints don’t produce any results.

Chapter 9: Persisting & Customizing Commands

As you've probably noticed in your development career, typing the same thing over and over really sucks. If you use a particular command that's difficult to type, there's no reason you should have to type the whole thing out. Just as you've learned when creating breakpoints using regular expressions, you'd go crazy typing out the full names of some of those Swift functions.

The same idea can be applied to any commands, settings, or code executed in LLDB. However, there's two problems that haven't been addressed until now: persisting your commands and creating shortcuts for them! Every time you run a new LLDB session, all your previous commands you've executed will vanish!

In this chapter, you'll learn how to persist these choices through the `.lldbinit` file. By persisting your choices and making convenience commands for yourself, your debugging sessions will run much more smoothly and efficiently. This is also an important concept because from here on out, you'll use the `.lldbinit` file on a regular basis.

Persisting... how?

Whenever LLDB is invoked, it searches several directories for special initialization files. If found, these files will be loaded into LLDB as soon as LLDB starts up but before LLDB has attached to the process (important to know if you're trying to execute arbitrary code in the init file).

You can use these files to specify settings or create custom commands to do your debugging bidding.

LLDB searches for an initialization file in the following places:

1. **~/.lldbinit-[context]** where `[context]` is **Xcode**, if you are debugging with Xcode, or **lldb** if you are using the command line incarnation of LLDB. For example, if you wanted commands that were only available in LLDB while debugging in the Terminal, you'd add content to `~/.lldbinit-lldb`, while if you wanted to have commands only available to Xcode you'd use `~/.lldbinit-Xcode`.
2. Next, LLDB searches for content found in **~/.lldbinit**. This is the ideal file for most of your logic, since you want to use commands in both Xcode and terminal sessions of LLDB.
3. Finally, LLDB will search the directory where it was invoked. Unfortunately, when Xcode launches LLDB, it'll launch LLDB at the `/` root directory. This isn't an ideal place to stick an `.lldbinit` file, so this particular implementation will be ignored throughout the book.

Creating the .lldbinit file

In this section you're going to create your first `.lldbinit` file.

First, open a Terminal window and type the following:

```
nano ~/.lldbinit
```

This uses the `nano` text editor to open up your `.lldbinit` file. If you already have an existing file in the location, `nano` will open up the file instead of creating a new one.

> **Note**: You really should be using some form of **vi** or **emacs** for editing .lldbinit, and then angrily blog about how unconventional the other editor is. I'm suggesting `nano` to stay out of the great debate.

Once the file is open in the `nano` editor, add the following line of code to the end of your `.lldbinit` file:

```
command alias -- Yay_Autolayout expression -l objc -O --
[[[[[UIApplication sharedApplication] keyWindow] rootViewController]
view] recursiveDescription]
```

You've just created an **alias** — a shortcut command for a longer expression. The alias's name is called `Yay_Autolayout` and it'll execute an expression command to get the root UIView (iOS only) and dump the position and layout of the root view and all of it's subviews.

Save your work by pressing **Ctrl + O**, but don't exit `nano` just yet.

Open the Signals Xcode project — you know, the one you've been playing with throughout this section. Build and run the Signals application. Once running, pause execution and type the alias in the debugger:

```
(lldb) Yay_Autolayout
```

This will dump out all the views in the applications! Neat!

> **Note**: The cool thing about this command is it'll work equally well for apps you do — and don't — have source code for. You could, hypothetically, attach LLDB to the Simulator's SpringBoard and dump all the views using the exact same method.

Now, use LLDB to get help for this new command:

```
(lldb) help Yay_Autolayout
```

The output will look kinda meh. You can do better. Go back to the `nano` Terminal window and rewrite the command alias to include some helpful information, like so:

```
command alias -H "Yay_Autolayout will get the root view and recursively
dump all the subviews and their frames" -h "Recursively dump views" --
Yay_Autolayout expression -l objc -O -- [[[[UIApplication
sharedApplication] keyWindow] rootViewController] view]
recursiveDescription]
```

Make sure `nano` saves the file by pressing **Ctrl + O**. Next, build and run the Signals project.

Now when you stop the debugger and type `help Yay_Autolayout`, you'll get help text at the bottom of the output. This is done with the `-H` command. You can also get a brief summary by just typing `help`, which gives the `-h` description along with the rest of the commands.

This may seem a bit pointless now, but when you have many, many custom commands in your `.lldbinit` file, you'll be thankful you provided documentation for yourself.

Command aliases with arguments

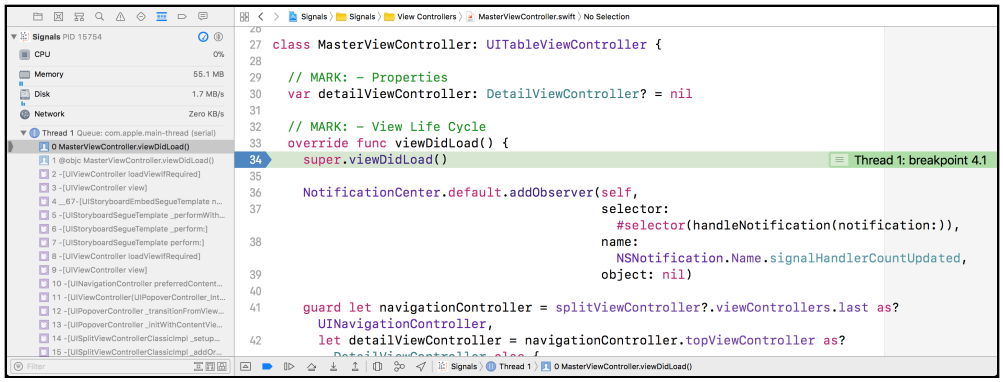
You've just created a standalone command alias that doesn't require any arguments. However, you'll often want to create aliases to which you can supply input.

Go back to the `nano` window in Terminal. Add the following at the bottom of the file:

```
command alias cpo expression -l objc -O --
```

You've just created a new command called **cpo**. The `cpo` command will do a normal `po` (print object), but it'll use the Objective-C context instead. This is an ideal command to use when you're in a Swift context, but want to use Objective-C to print out an address or register of something you know is a valid Objective-C object.

Save your work in nano, and jump over to the Signals project. Navigate to **MasterViewController**'s `viewDidLoad` and set a breakpoint at the top of the function. Build and run the application.



To best understand the importance of the `cpo` command, first get the reference to the MasterViewController.

```
(lldb) po self
```

You'll get output similar to the following:

```
<Signals.MasterViewController: 0x7fc8295071a0>
```

Take the memory address you get at the end of the output (as usual, yours will likely be different), and try printing that in the debugger.

```
(lldb) po 0x7fc8295071a0
```

This will not produce any meaningful output, since you've stopped in a Swift file, and Swift is a type-safe language. Simply printing an address in Swift will not do anything.

This is why the Objective-C context is so useful when debugging, especially when working in assembly where there are only references to memory addresses.

Now, use the new command you've just created on the address:

```
(lldb) cpo 0x7fc8295071a0
```

You'll see the same output as you did with `po self`:

```
<Signals.MasterViewController: 0x7fc8295071a0>
```

This is a helpful command to get a NSObject's description, whether it's created with Objective-C or Swift.

Where to go from here?

You've learned how to create aliases for simple commands as well as persist them in the `.lldbinit` file. This will work across both Xcode and Terminal invocations of LLDB.

As an exercise, add help messages to your newly created `cpo` command in the `~/.lldbinit` file so you'll be able to remember how to use it when you have an onslaught of custom commands. Remember the `-h` option is the short help message that's displayed when you just type `help`, while the `-H` option is the longer help command used when you type `help command`. Remember to use the `--` to separate your help input arguments to the rest of your command.

In addition, write a command alias for something you often use. Put this alias in your `~/.lldbinit` file and try it out!

Chapter 10: Regex Commands

In the previous chapter, you learned about the `command alias` command as well as how to persist commands through the `lldbinit` file. Unfortunately, `command alias` has some limitations.

An alias created this way will work great if you're trying to execute a static command, but usually you'd want to feed input into a command in order to get some useful output.

Where `command alias` falls short is it essentially *replaces* the alias with the actual command. What if you wanted to have input supplied into the middle of a command, such as a command to get the class of a given object instance, providing the object as an input?

Fortunately, there *is* an elegant solution to supplying input into a custom LLDB command using a `command regex`.

command regex

The LLDB command **`command regex`** acts much like `command alias`, except you can provide a regular expression for input which will be parsed and applied to the action part of the command.

`command regex` takes an input syntax that looks similar to the following:

```
s/<regex>/<subst>/
```

This is a normal regular expression. It starts with '`s/`', which specifies a stream editor input to use the **substitute command**. The `<regex>` part is the bit that specifies what should be replaced. The `<subst>` part says what to replace it with.

> **Note**: This syntax is derived from the **sed** Terminal command. This is important to know, because if you're experimenting using advanced patterns, you can check the **man** pages of `sed` to see what's possible within the substitute formatting syntax.

Time to look at a concrete example. Open up the **Signals** Xcode project. Build and run, then pause the application in the debugger. Once the LLDB console is up and ready to receive input, enter the following command in LLDB:

```
(lldb) command regex rlook 's/(.+)/image lookup -rn %1/'
```

This command you've entered will make your image regex searches much easier. You've created a new command called `rlook`. This new command takes everything after the `rlook` and prefixes it with `image lookup -rn`. It does this through a regex with a single matcher (the parentheses) which matches on one or more characters, and replaces the whole thing with `image lookup -rn %1`. The `%1` specifies the contents of the matcher.

So, for example, if you enter this:

```
rlook FOO
```

LLDB will actually execute the following:

```
image lookup -rn FOO
```

Now, instead of having to type the soul-crushingly long `image lookup -rn`, you can just type `rlook`!

But wait, it gets better. Provided there are no conflicts with the characters `rl`, you can simply use that instead. You can specify any command, be it built-in or your own, by using any prefix which is not shared with another command.

This means you can easily search for methods like `viewDidLoad` using a much more convenient amount of typing. Try it out now:

```
(lldb) rl viewDidLoad
```

This will produce all the `viewDidLoad` implementations across all modules in the current executable. Try limiting it to only code in the Signals app:

```
(lldb) rl viewDidLoad Signals
```

Now you're satisfied with the command, add the following line of code to your `~/.lldbinit` file:

```
command regex rlook 's/(.+)/image lookup -rn %1/'
```

Note: The best way to implement a regex command is to use LLDB while a program is running. This lets you iterate on the command regex (by redeclaring it if you're not happy with it) and test it out without having to relaunch LLDB. Once you're happy with the command, add it to your ~/.lldbinit file so it will be available every time LLDB starts up. Now the `rlook` command will be available to you from here on out, resulting in no more painful typing of the full `image lookup -rn` command. Yay!

Executing complex logic

Time to take the `command regex` up a level. You can actually use this command to execute multiple commands for a single alias. While LLDB is still paused, implement this new command:

```
(lldb) command regex -- tv 's/(.+)/expression -l objc -O -- @import
QuartzCore; [%1 setHidden:!(BOOL)[%1 isHidden]]; (void)[CATransaction
flush];/'
```

This complicated, yet useful command, will create a command named **`tv`** (toggle view), which toggles a UIView (or NSView) on or off while the debugger is paused.

Packed into this command are three separate lines of code:

1. `@import QuartzCore` imports the QuartzCore framework into the debugger's address space. This is required because the debugger won't understand what code you're executing until it's declared. You're about to execute code from the QuartzCore framework, so just in case it hasn't been imported yet, you're doing it now.

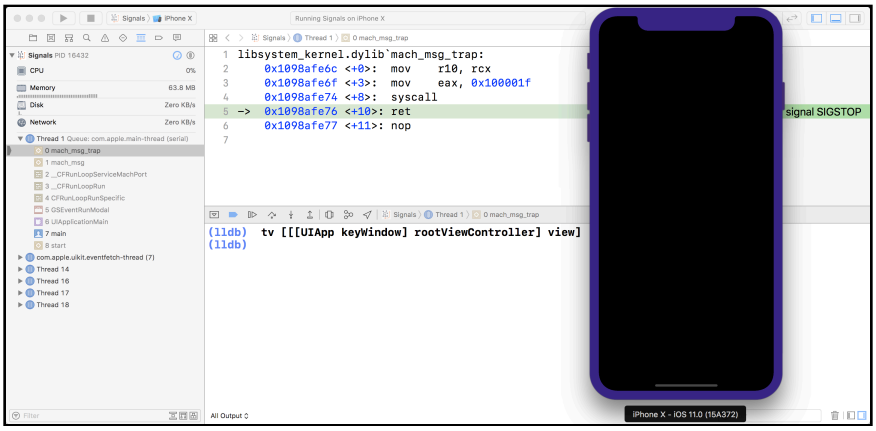
2. `[%1 setHidden:!(BOOL)[%1 isHidden]];` toggles the view to either hidden or visible, depending what the previous state was. Note that `isHidden` doesn't know the return type, so you need to cast it to an Objective-C `BOOL`

3. The final command, `[CATransaction flush]`, will flush the CATransaction queue. Manipulating the UI in the debugger will normally mean the screen will not reflect any updates until the debugger resumes execution. However, this method will update the screen resulting in LLDB not needing to `continue` in order to show visual changes.

> **Note**: Due to the limitations of the input params, specifying multiline input is not allowed so you have to join all the commands onto one line. This is ugly but necessary when crafting these regex commands. However, if you ever do this in actual Objective-C/Swift source code, may the Apple Gods punish you with extra-long app review times!

Provided LLDB is still paused, execute this newly created `tv` command:

```
(lldb) tv [[[UIApp keyWindow] rootViewController] view]
```



Bring up the Simulator to verify the view has disappeared.

Now simply press **Enter** in the LLDB console, as LLDB will repeat the last command you've entered. The view will flash back to normal.

Now that you're done implementing the `tv` command, add it to your `~/.lldbinit` file:

```
command regex -- tv 's/(.+)/expression -l objc -O -- @import QuartzCore;
[%1 setHidden:!(BOOL)[%1 isHidden]]; (void)[CATransaction flush];/'
```

Chaining regex inputs

There's a reason why that weird stream editor input was chosen for using this command: this format lets you easily specify multiple actions for the same command. When given multiple commands, the regex will try to match each input. If the input matches, that particular `<subst>` is applied to the command. If the input doesn't match for a particular stream, it'll go to the next command and see if the regex can match that input.

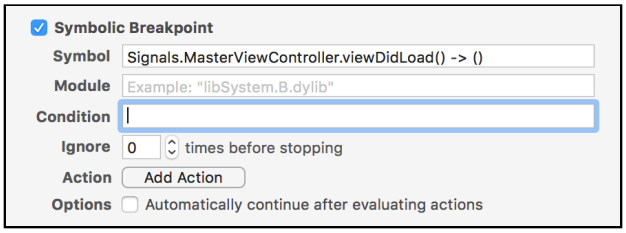
It's generally necessary to use the Objective-C context when working with objects in memory and registers. Also, anything that begins with the square open bracket or the '@' character is (likely) Objective-C. This is because Swift makes it difficult to work with memory, and it won't let you access registers, nor do Swift expressions usually ever begin with an open bracket or '@' character.

You can use this information to automatically detect which context you need to use for a given input.

Let's see how you'd you go about building a command which gets the class information out of an object, which honors the above requirements.

- In Objective-C, you'd use **`[objcObject class]`**.
- In Swift, you'd use **`type(of: swiftObject)`**.

In Xcode, create a Symbolic breakpoint on `Signals.MasterViewController.viewDidLoad() -> ()` (make sure to keep the spacing).



Build and run, then wait for the breakpoint to be triggered. As usual, head on over to the debugger.

First, build out the Objective-C implementation of this new command, **`getcls`**.

```
(lldb) command regex getcls 's/(([0-9]|\$|\@|\[).*)/cpo [%1 class]/'
```

Wow, that regex makes the eyes blur. Time to break it down:

At first, there's an inner grouping saying the following characters can be used to match the start:

- `[0-9]` means the numbers from 0-9 can be used.
- `\$` means the literal character '$' will be matched
- `\@` means the literal character '@' will be matched
- `\[` means the literal character '[' will be matched

Any characters that start with the above will generate a match. Following that is `.*` which means zero or more characters will produce a match.

Overall, this means that a number, `$`, `@`, or `[`, followed by any characters will result in the command matching and running `cpo [%1 class]`. Once again, `%1` is replaced with the first matcher from the regex. In this case, it's the entire command. The inner matcher (matching a number, `$`, or so on) would be `%2`.

Try throwing a couple of commands at the `getcls` command to see how it works:

```
(lldb) getcls @"hello world"
__NSCFString

(lldb) getcls @[@"hello world"]
__NSSingleObjectArrayI

(lldb) getcls [UIDevice currentDevice]
UIDevice

(lldb) cpo [UIDevice currentDevice]
<UIDevice: 0x60800002b520>

(lldb) getcls 0x60800002b520
UIDevice
```

Awesome!

However, this only handles references that make sense in the Objective-C context and that match your command. For example, try the following:

```
(lldb) getcls self
```

You'll get an error:

```
error: Command contents 'self' failed to match any regular expression in
the 'getcls' regex command.
```

This is because there was no matching regex for the input you provided. Let's add one which catches other forms of input to `getcls`. Type the following into LLDB now:

```
(lldb) command regex getcls 's/(([0-9]|\$|\@|\[).*)/cpo [%1 class]/' 's/
(.+)/expression -l swift -O -- type(of: %1)/'
```

This looks a bit more complex, but it's not too bad. The first part of the command is the same as you added before. But now you've added another regex to the end. This one is a catch-all, just like the `rlook` command you added. This catch-all simply calls `type(of:)` with the input as the parameter.

Try executing the command again for `self`:

```
(lldb) getcls self
```

You'll now get the expected `Signals.MasterViewController` output. Since you made the Swift context as a catch-all, you can use this command in interesting ways.

```
(lldb) getcls self .title
```

Notice the space in there, and it still works. This is because you told the Swift context to quite literally take anything except newlines.

Once, you're done playing with this new and improved `getcls` command, be sure to add it to your `~/.lldbinit` file.

Supplying multiple parameters

The final party trick you'll explore in `command regex` is supplying multiple parameters to `command regex`. However, before you do that, revisit the first command:

```
(lldb) command regex rlook 's/(.+)/image lookup -rn %1/'
```

Take a look at the `(.+)`. The parentheses around this make it what is known as a **capture group**. The `%1` in the right hand side of the substitution (the replacement) indicates that the `%1` should be replaced with the first capture group. Therefore this whole regular expression means that the entire text is captured, and `image lookup -rn` is added before it.

By supplying more capture groups, you can add more parameters to parse.

In an earlier chapter, you explored the private `statusBar` property of the `UIApplication` instance.

You can look it up using Objective-C like so:

```
(lldb) ex -l objc -O -- [[UIApplication shared] statusBar]
```

But this is a private API meaning you can't call it via Swift unless you bring the Objective-C runtime into play, or add `statusBar` method in a category in your own header.

To be able to execute this in the Swift LLDB context, you can execute the following:

```
(lldb) ex -l swift -O --
UIApplication.shared.perform(NSSelectorFromString("statusBar"))
```

This is a little much to type. You can use the command regex to create multiple capture groups for this command.

In LLDB, type the following:

```
(lldb) command regex swiftperfsel 's/(.+)\s+(\w+)/expression -l swift -O
-- %1.perform(NSSelectorFromString("%2"))/'
```

And then give this command a whirl:

```
(lldb) swiftperfsel UIApplication.shared statusBar
```

You'll get the Swift formatted output to this private property:

```
▿ Optional<Unmanaged<AnyObject>>
  ▿ some : Unmanaged<AnyObject>
    - _value : <UIStatusBar_Modern: 0x7fa86dc05820; frame = (0 0; 375
44); autoresize = W+BM; layer = <CALayer: 0x6000018e5000>>
```

As you can see, adding multiple parameters quickly ups the complexity of the regular expression. By combining multiple capture groups with multiple chained groups, you can make a rather versatile command regex to handle all types of optional and required input. However, that's going to look really, really ugly since all of this needs to be declared on one line.

Fortunately, LLDB has the **script bridging** interface — a fully featured Python implementation for creating advanced LLDB commands to do your debugging bidding. You'll take an in depth look at script bridging in the 4th section of this book.

For now, simply use either `command alias` or `command regex` to suit your debugging needs.

Where to go from here?

Go back to the regex commands you've created in this chapter and add **syntax** and **help** help documentation.

You'll thank yourself for this documentation about your command's functionality, when it's 11 PM on a Friday night and you just want to figure out this gosh darn bug.

Section II: Understanding Assembly

Knowing what the computer is doing with all those 1s and 0s underneath your code is an excellent skill to have when digging for useful information about a program. This section will set you up with the theory you'll need for the remainder of this book in order to create complex debugging scripts — and introduce you to the basic theory behind reverse-engineering code.

Chapter 11: Assembly Register Calling Convention

Chapter 12: Assembly & Memory

Chapter 13: Assembly & the Stack

Chapter 11: Assembly Register Calling Convention

Now that you've gained a basic understanding of how to maneuver around the debugger, it's time to take a step down the executable Jenga tower and explore the 1s and 0s that make up your source code. This section will focus on the low-level aspects of debugging.

In this chapter, you'll look at registers the CPU uses and explore and modify parameters passed into function calls. You'll also learn about common Apple computer architectures and how their registers are used within a function. This is known as an architecture's **calling convention**.

Knowing how assembly works and how a specific architecture's calling convention works is an extremely important skill to have. It lets you observe function parameters you don't have the source code for and lets you modify the parameters passed into a function. In addition, it's sometimes even better to go to the assembly level because your source code could have different or unknown names for variables you're not aware of.

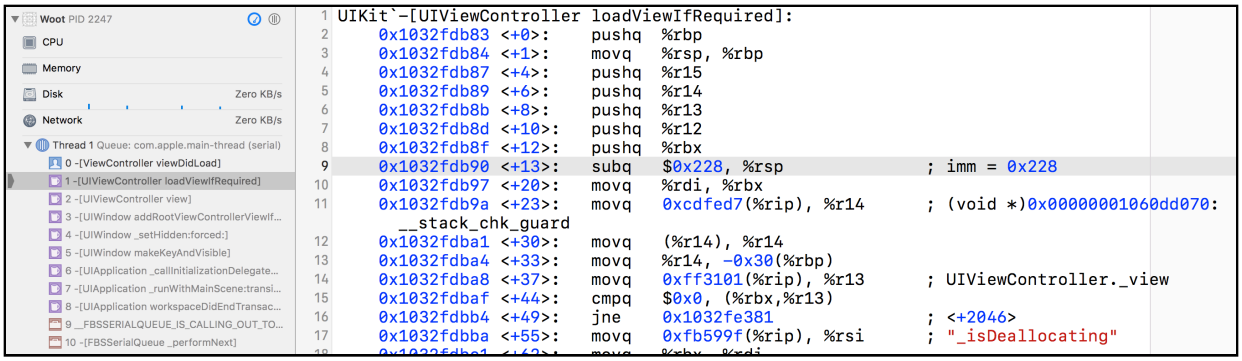
For example, let's say you always wanted to know the second parameter of a function call, regardless of what the parameter's name is. Knowledge of assembly gives you a great base layer to manipulate and observe parameters in functions.

Assembly 101

Wait, so what's assembly again?

Have you ever stopped in a function you didn't have source code for, and saw an onslaught of memory addresses followed by scary, short commands? Did you huddle in a ball and quietly whisper to yourself you'll never look at this dense stuff again? Well... that stuff is known as assembly!

Here's a picture of a backtrace in Xcode, which showcases the assembly of a function within the Simulator.



Looking at the image above, the assembly can be broken into several parts. Each line in a assembly instruction contains an **opcode**, which can be thought of as an extremely simple instruction for the computer.

So what does an opcode look like? An opcode is an instruction that performs a simple task on the computer. For example, consider the following snippet of assembly:

```
pushq   %rbx
subq    $0x228, %rsp
movq    %rdi, %rbx
```

In this block of assembly, you see three opcodes, **`pushq`**, **`subq`**, and **`movq`**. Think of the opcode items as the action to perform. The things following the opcode are the source and destination labels. That is, these are the items the opcode acts upon.

In the above example, there are several **registers**, shown as **`rbx`**, **`rsp`**, **`rdi`**, and **`rbp`**. The **`%`** before each tells you this is a register.

In addition, you can also find a numeric constant in hexadecimal shown as **`0x228`**. The **`$`** before this constant tells you it's an absolute number.

There's no need to know what this code is doing at the moment, since you'll first need to learn about the registers and calling convention of functions. Then you'll learn more about the opcodes and write your own assembly in a future chapter.

> **Note**: In the above example, take note there are a bunch of `%`'s and `$`'s that precede the registers and constants. This is how the disassembler formats the assembly. However, there are two main ways that assembly can be showcased. The first is **Intel** assembly, and the second is **AT&T** assembly.
>
> By default, Apple's disassembler tools ship with assembly displayed in the AT&T format, as it is in the example above. Although this is a good format to work with, it can be a little hard on the eyes. In the next chapter, you'll change the assembly format to Intel, and will work exclusively with Intel assembly syntax from there on out.

x86_64 vs ARM64

As a developer for Apple platforms, there are two primary architectures you'll deal with when learning assembly: **x86_64** architecture and **ARM64** architecture. x86_64 is the architecture most likely used on your macOS computer, unless you are running an "ancient" Macintosh.

x86_64 is a **64-bit** architecture, which means every address can hold up to 64 1s or 0s. Alternatively, older Macs use a **32-bit** architecture, but Apple stopped making 32-bit Macs at the end of the 2010's. Programs running under macOS are likely to be 64-bit compatible, including programs on the Simulator. That being said, even if your macOS is x86_64, it can still run 32-bit programs.

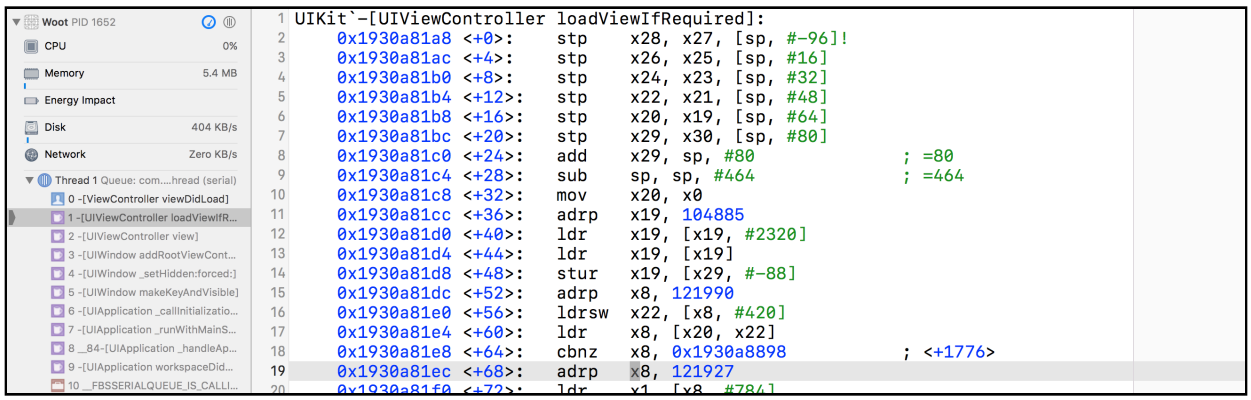
If you have any doubt of what hardware architecture you're working with, you can get your computer's hardware architecture by running the following command in Terminal:

```
uname -m
```

ARM64 architecture is used on mobile devices such as your iPhone where limiting energy consumption is critical.

ARM emphasizes power conservation, so it has a reduced set of opcodes that help facilitate energy consumption over complex assembly instructions. This is good news for you, because there are fewer instructions for you to learn on the ARM architecture.

Here's a screenshot of the same method shown earlier, except this time in ARM64 assembly on an iPhone 7:



You might not be able to differentiate between the two architectures now, but you'll soon know them like the back of your hand.

Apple originally shipped 32-bit ARM processors in many of their devices, but have since moved to 64-bit ARM processors. 32-bit devices are almost obsolete as Apple has phased them out through various iOS versions. For example, the iPhone 5 is a 32-bit device which is not supported in iOS 12. However, the iPhone 5s is a 64-bit device which is supported in iOS 12.

Since it's best to focus on what you'll need for the future, this book will focus primarily on 64-bit assembly for both architectures. In addition, you'll start learning x86_64 assembly first and then transition to learning ARM64 assembly so you don't get confused. Well, not *too* confused.

x86_64 register calling convention

Your CPU uses a set of registers in order to manipulate data in your running program. These are storage holders, just like the RAM in your computer. However they're located on the CPU itself very close to the parts of the CPU that need them. So these parts of the CPU can access these registers incredibly quickly.

Most instructions involve one or more registers and perform operations such as writing the contents of a register to memory, reading the contents of memory to a register or performing arithmetic operations (add, subtract, etc.) on two registers.

In **x64** (from here on out, x64 is an abbreviation for x86_64), there are **16 general purpose registers** used by the machine to manipulate data.

These registers are **RAX**, **RBX**, **RCX**, **RDX**, **RDI**, **RSI**, **RSP**, **RBP** and **R8** through **R15**.

These names will not mean much to you now, but you'll explore the importance of each register soon.

When you call a function in x64, the manner and use of the registers follows a very specific convention. This dictates where the parameters to the function should go and where the return value from the function will be when the function finishes. This is important so code compiled with one compiler can be used with code compiled with another compiler. Swift is a bit like the wild west when it comes to calling conventions, so you'll start by learning through Objective-C and you'll move to learning about Swift later.

Take a look at this simple Objective-C code:

```
NSString *name = @"Zoltan";
NSLog(@"Hello world, I am %@. I'm %d, and I live in %@.", name, 30, @"my
father's basement");
```

There are four parameters passed into the `NSLog` function call. Some of these values are passed as-is, while one parameter is stored in a local variable, then referenced as a parameter in the function. However, when viewing code through assembly, the computer doesn't care about names for variables; it only cares about locations in memory.

The following registers are used as parameters when a function is called in x64 assembly. Try and commit these to memory, as you'll use these frequently in the future:

- First Argument: `RDI`
- Second Argument: `RSI`
- Third Argument: `RDX`
- Fourth Argument: `RCX`
- Fifth Argument: `R8`
- Sixth Argument: `R9`

If there are more than six parameters, then the program's stack is used to pass in additional parameters to the function.

Going back to that simple Objective-C code, you can re-imagine the registers being passed like the following pseudo-code:

```
RDI = @"Hello world, I am %@. I'm %d, and I live in %@.";
RSI = @"Zoltan";
RDX = 30;
```

```
RCX = @"my father's basement";
NSLog(RDI, RSI, RDX, RCX);
```

As soon as the `NSLog` function starts, the given registers will contain the appropriate values as shown above.

However, as soon as the **function prologue** (the beginning section of a function that prepares the stack and registers) finishes executing, the values in these registers will likely change. The generated assembly will likely overwrite the values stored in these registers, or just simply discard these references when the code has no more need of them.

This means as soon as you leave the start of a function (through stepping over, stepping in, or stepping out), you can no longer assume these registers will hold the expected values you want to observe, unless you actually look at the assembly code to see what it's doing.

This calling convention heavily influences your debugging (and breakpoint) strategy. If you were to automate any type of breaking and exploring, you would have to stop at the start of a function call in order to inspect or modify the parameters without having to actually dive into the assembly.

Objective-C and registers

As you learned in the previous section, registers use a specific calling convention. You can take that same knowledge and apply it to other languages as well.

When Objective-C executes a method, a special C function, **`objc_msgSend`**, is executed. There's actually several different types of these functions, but `objc_msgSend` is the most widely used, as this is the heart of message dispatch. As the first parameter, `objc_msgSend` takes the reference of the object upon which the message is being sent. This is followed by a **selector**, which is simply just a `char *` specifying the name of the method being called on the object. Finally, `objc_msgSend` takes a variable amount of arguments within the function if the Selector specifies there should be parameters.

Let's look at a concrete example of this in an iOS context:

```
[UIApplication sharedApplication];
```

The compiler will take this code and create the following pseudocode:

```
id UIApplicationClass = [UIApplication class];
objc_msgSend(UIApplicationClass, "sharedApplication");
```

The first parameter is a reference to the `UIApplication` class, followed by the `sharedApplication` selector. An easy way to tell if there are any parameters is to simply check for colons in the Objective-C Selector. Each colon will represent a parameter in a Selector.

Here's another Objective-C example:

```
NSString *helloWorldString = [@"Can't Sleep; "
stringByAppendingString:@"Clowns will eat me"];
```

The compiler will create the following (shown below in pseudocode):

```
NSString *helloWorldString;
helloWorldString = objc_msgSend(@"Can't Sleep; ",
"stringByAppendingString:", @"Clowns will eat me");
```

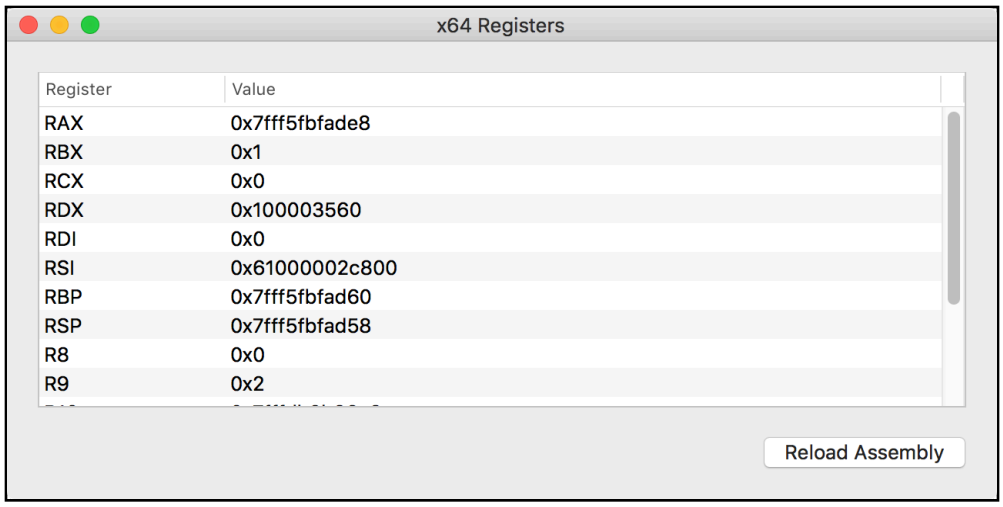
The first argument is an instance of an `NSString` (`@"Can't Sleep; "`), followed by the Selector, followed by a parameter which is also an `NSString` instance.

Using this knowledge of `objc_msgSend`, you can use the registers in x64 to help explore content, which you'll do very shortly.

Putting theory to practice

For this section, you'll be using a project supplied in this chapter's resource bundle called **Registers**.

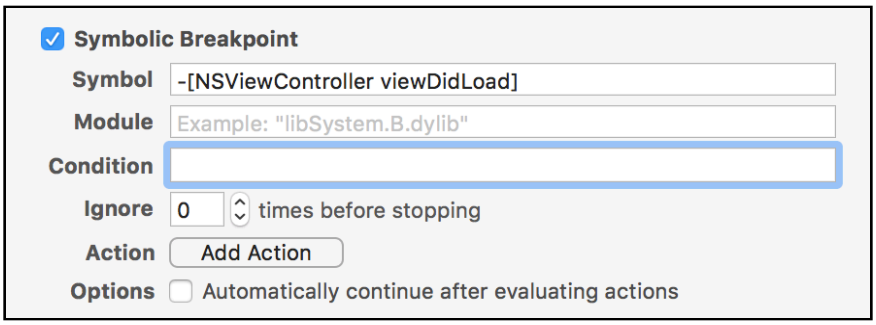
Open this project up through Xcode and give it a run.



This is a rather simple application which merely displays the contents of some x64 registers. It's important to note that this application can't display the values of

registers at any given moment; it can only display the values of registers during a specific function call. This means that you won't see too many changes to the values of these registers since they'll likely have the same (or similar) value when the function to grab the register values is called.

Now that you've got an understanding of the functionality behind the Registers macOS application, create a symbolic breakpoint for `NSViewController`'s `viewDidLoad` method. Remember to use "NS" instead of "UI", since you're working on a Cocoa application.



Build and rerun the application. Once the debugger has stopped, type the following into the LLDB console:

```
(lldb) register read
```

This will list all of the main registers at the paused state of execution. However, this is too much information. You should selectively print out registers and treat them as Objective-C objects instead.

If you recall, `-[NSViewController viewDidLoad]` will be translated into the following assembly pseudocode:

```
RDI = UIViewControllerInstance
RSI = "viewDidLoad"
objc_msgSend(RDI, RSI)
```

With the x64 calling convention in mind, and knowing how `objc_msgSend` works, you can find the specific `NSViewController` that is being loaded.

Type the following into the LLDB console:

```
(lldb) po $rdi
```

You'll get output similar to the following:

```
<Registers.ViewController: 0x6080000c13b0>
```

This will dump out the `NSViewController` reference held in the `RDI` register, which as you now know, is the location of the first argument to the method.

In LLDB, it's important to prefix registers with the `$` character, so LLDB knows you want the value of a register and not a variable related to your scope in the source code. Yes, that's different than the assembly you see in the disassembly view! Annoying, eh?

> **Note**: The observant among you might notice whenever you stop on an Objective-C method, you'll never see the `objc_msgSend` in the LLDB backtrace. This is because the `objc_msgSend` family of functions perfoms a **`jmp`**, or jump opcode command in assembly. This means that `objc_msgSend` acts as a trampoline function, and once the Objective-C code starts executing, all stack trace history of `objc_msgSend` will be gone. This is an optimization known as **tail call optimization**.

Try printing out the `RSI` register, which will hopefully contain the Selector that was called. Type the following into the LLDB console:

```
(lldb) po $rsi
```

Unfortunately, you'll get garbage output that looks something like this:

```
140735181830794
```

Why is this?

An Objective-C Selector is basically just a `char *`. This means, like all C types, LLDB does not know how to format this data. As a result, you must explicitly cast this reference to the data type you want.

Try casting it to the correct type:

```
(lldb) po (char *)$rsi
```

You'll now get the expected:

```
"viewDidLoad"
```

Of course, you can also cast it to the Selector type to produce the same result:

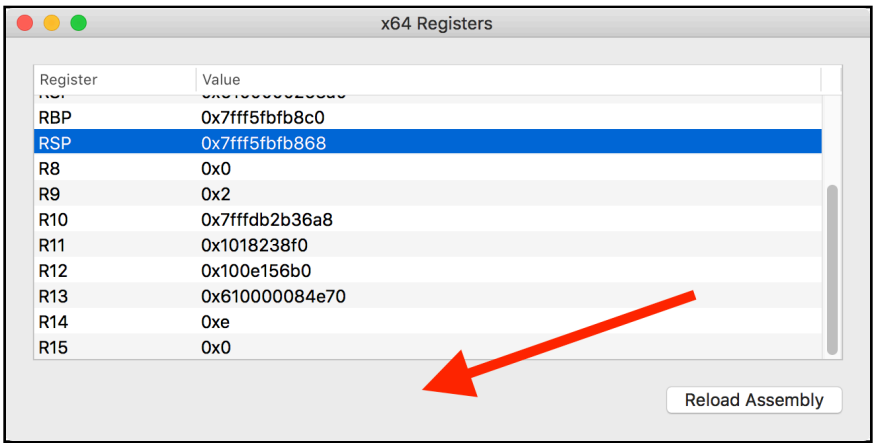
```
(lldb) po (SEL)$rsi
```

Now, it's time to explore an Objective-C method with arguments. Since you've stopped on `viewDidLoad`, you can safely assume the `NSView` instance has loaded. A method of interest is the **`mouseUp:`** Selector implemented by `NSView`'s parent class, `NSResponder`.

In LLDB, create a breakpoint on `NSResponder`'s `mouseUp:` Selector and resume execution. If you can't remember how to do that, here are the commands you need:

```
(lldb) b -[NSResponder mouseUp:]
(lldb) continue
```

Now, click on the application's window. Make sure to click on the outside of the NSScrollView as it will gobble up your click and the `-[NSResponder mouseUp:]` breakpoint will not get hit.



As soon as you let go of the mouse or the trackpad, LLDB will stop on the `mouseUp:` breakpoint. Print out the reference of the `NSResponder` by typing the following into the LLDB console:

```
(lldb) po $rdi
```

You'll get something similar to the following:

```
<NSView: 0x608000120140>
```

However, there's something interesting with the Selector. There's a colon in it, meaning there's an argument to explore! Type the following into the LLDB console:

```
(lldb) po $rdx
```

You'll get the description of the NSEvent:

```
NSEvent: type=LMouseUp loc=(351.672,137.914) time=175929.4 flags=0
win=0x6100001e0400 winNum=8622 ctxt=0x0 evNum=10956 click=1
buttonNumber=0 pressure=0 deviceID:0x300000014400000
subtype=NSEventSubtypeTouch
```

How can you tell it's an `NSEvent`? Well, you can either look online for documentation on `-[NSResponder mouseUp:]` or, you can simply use Objective-C to get the type:

```
(lldb) po [$rdx class]
```

Pretty cool, eh?

Sometimes it's useful to use registers and breakpoints in order to get a reference to an object you know is alive in memory.

For example, what if you wanted to change the front NSWindow to red, but you had no reference to this view in your code, and you didn't want to recompile with any code changes? You can simply create a breakpoint you can easily trip, get the reference from the register and manipulate the instance of the object as you please. You'll try changing the main window to red now.

> **Note**: Even though `NSResponder` implements `mouseDown:`, `NSWindow` overrides this method since it's a subclass of `NSResponder`. You can dump all classes that implement `mouseDown:` and figure out which of those classes inherit from `NSResponder` to determine if the method is overridden without having access to the source code. An example of dumping all the Objective-C classes that implement `mouseDown:` is `image lookup -rn '\ mouseDown:'`

First remove any previous breakpoints using the LLDB console:

```
(lldb) breakpoint delete
About to delete all breakpoints, do you want to do that?: [Y/n]
```

Then type the following into the LLDB console:

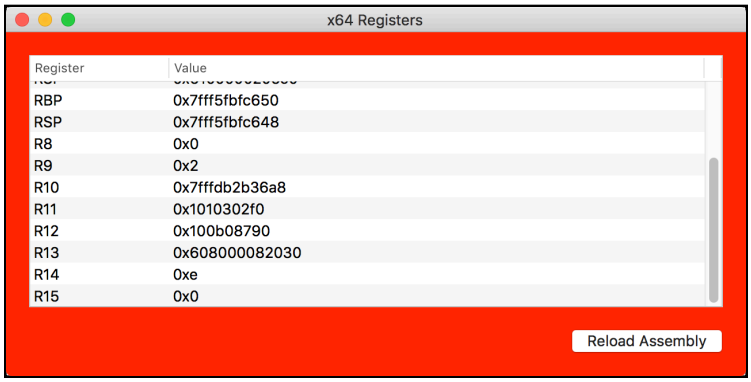
```
(lldb) breakpoint set -o true -S "-[NSWindow mouseDown:]"
(lldb) continue
```

This sets a breakpoint which will fire only once — a one-shot breakpoint.

Tap on the application. Immediately after tapping, the breakpoint should trip. Then type the following into the LLDB console:

```
(lldb) po [$rdi setBackgroundColor:[NSColor redColor]]
(lldb) continue
```

Upon resuming, the NSWindow will change to red!



Swift and registers

When exploring registers in Swift you'll hit three hurdles that make assembly debugging harder than it is in Objective-C.

1. First, registers are **not** available in the Swift debugging context. This means you have to get whatever data you want and then use the Objective-C debugging context to print out the registers passed into the Swift function. Remember that you can use the `expression -l objc -O --` command, or alternatively use the `cpo` custom command you made in Chapter 9, "Persisting and Customizing Commands". Fortunately, the `register read` command is available in the Swift context.

2. Second, Swift is not as dynamic as Objective-C. In fact, it's sometimes best to assume that Swift is like C, except with a very, very cranky and bossy compiler. If you have a memory address, you need to explicitly cast it to the object you expect it to be; otherwise, the Swift debugging context has no clue how to interpret a memory address.

3. As of Swift 4.2, the calling convention is still not stabilized! That means that Swift calling into C or Objective-C code can reliably work (since they have, like, standards, omg...), but C/Objective-C code can't reliably call into Swift code. This has resulted in the "crowd favorite", Swift Bridging Headers. What's even more depressing is you can't confidentatlly rely on the registers being used in the same manner across different versions of Swift!

When Swift calls a function, it has no need to use `objc_msgSend`, unless you mark up a method to use **`@objc`**. In addition, the latest 4.2 version of Swift will oftentimes opt to remove the `self` register (RDI) as the first parameter and instead place it on the stack. This means that the RDI register, which originally held the instance to `self`, and the RSI

register, which originally held the Selector in Objective-C, are freed up to handle parameters for a function. This is done in the name of "optimization", but the compiler's inconsistency results in incompatible code and tools which struggle to analyze Swift generated assembly. It has also resulted in version updates for this book to be a major PITA, since the Swift authors seem to come up with a new calling convention each year for Swift.

Enough theory — time to see how this year's Swift 4.2 version will compile code.

In the Registers project, navigate to **ViewController.swift** and add the following function below `viewDidLoad`:

```
func executeLotsOfArguments(one: Int, two: Int, three: Int,
                            four: Int, five: Int, six: Int,
                            seven: Int, eight: Int, nine: Int,
                            ten: Int) {
    print("arguments are: \(one), \(two), \(three),
          \(four), \(five), \(six), \(seven),
          \(eight), \(nine), \(ten)")
}
```

Next, add the following to the end of `viewDidLoad` to call this new function with the appropriate arguments:

```
self.executeLotsOfArguments(one: 1, two: 2, three: 3, four: 4,
                            five: 5, six: 6, seven: 7,
                            eight: 8, nine: 9, ten: 10)
```

Put a breakpoint on the very same line as of the declaration of `executeLotsOfArguments` so the debugger will stop at the very beginning of the function. This is important, or else the registers might get clobbered if the function is actually executing.

Finally, remove the symbolic breakpoint you set on `-[NSViewController viewDidLoad]`.

Build and run the app, then wait for the `executeLotsOfArguments` breakpoint to stop execution.

Again, a good way to start investigating is to dump the list registers. In LLDB, type the following:

```
(lldb) register read -f d
```

This will dump the registers and display the format in decimal by using the `-f d` option.

The output will look similar to the following:

```
General Purpose Registers:
       rax = 106377750908800
       rbx = 106377750908800
       rcx = 4
       rdx = 3
       rdi = 1
       rsi = 2
       rbp = 140732920750896
       rsp = 140732920750776
        r8 = 5
        r9 = 6
       r10 = 4294981504
Registers`Registers.ViewController.executeLotsOfArguments(one: Swift.Int,
two: Swift.Int, three: Swift.Int, four: Swift.Int, five: Swift.Int, six:
Swift.Int, seven: Swift.Int, eight: Swift.Int, nine: Swift.Int, ten:
Swift.Int) -> Swift.String at ViewController.swift:39
       r11 = 105827995224480
       r12 = 4314945952
       r13 = 106377750908800
       r14 = 88
       r15 = 4314945952
       rip = 4294981504
Registers`Registers.ViewController.executeLotsOfArguments(one: Swift.Int,
two: Swift.Int, three: Swift.Int, four: Swift.Int, five: Swift.Int, six:
Swift.Int, seven: Swift.Int, eight: Swift.Int, nine: Swift.Int, ten:
Swift.Int) -> Swift.String at ViewController.swift:39
    rflags = 518
        cs = 43
        fs = 0
        gs = 0
```

As you can see, the registers follow the x64 calling convention. `RDI`, `RSI`, `RDX`, `RCX`, `R8` and `R9` hold your first six parameters.

You may (or may not depending on the Swift version) also notice other parameters are stored in some of the other registers. While this is true, it's simply a leftover from the code that sets up the stack for the remaining parameters. Remember, parameters after the sixth one go on the stack.

RAX, the return register

But wait — there's more! So far, you've learned how six registers are called in a function, but what about return values?

Fortunately, there is only one designated register for return values from functions: **RAX**. Go back to `executeLotsOfArguments` and modify the function to return an `Int`, like so:

```
func executeLotsOfArguments(one: Int, two: Int, three: Int,
                            four: Int, five: Int, six: Int,
                            seven: Int, eight: Int, nine: Int,
                            ten: Int) -> Int {
    print("arguments are: \(one), \(two), \(three), \(four),
          \(five), \(six), \(seven), \(eight), \(nine), \(ten)")
    return 100
}
```

In `viewDidLoad`, modify the function call to receive and ignore the String value.

```
override func viewDidLoad() {
    super.viewDidLoad()
    _ = self.executeLotsOfArguments(one: 1, two: 2,
          three: 3, four: 4, five: 5, six: 6, seven: 7,
          eight: 8, nine: 9, ten: 10)
}
```

Create a breakpoint somewhere in `executeLotsOfArguments`. Build and run again, and wait for execution to stop in the function. Next, type the following into the LLDB console:

```
(lldb) finish
```

This will finish executing the current function and pause the debugger again. At this point, the return value from the function should be in `RAX`. Type the following into LLDB:

```
(lldb) re re rax -fd
```

You'll get something similar to the following:

```
      rax = 100
```

Boom! Your return value!

Knowledge of the return value in `RAX` is extremely important as it will form the foundation of debugging scripts you'll write in later sections.

Changing around values in registers

In order to solidify your understanding of registers, you'll modify registers in an already-compiled application.

Close Xcode and the Registers project. Open a Terminal window and launch the iPhone X Simulator. Do this by typing the following:

```
xcrun simctl list
```

You'll see a long list of devices. Search for the latest iOS version for which you have a simulator installed. Underneath that section, find the iPhone X device. It will look something like this:

```
iPhone X (DE1F3042-4033-4A69-B0BF-FD71713CFBF6) (Shutdown)
```

The UUID is what you're after. Use that to open the iOS Simulator by typing the following, replacing your UUID as appropriate:

```
open /Applications/Xcode.app/Contents/Developer/Applications/
Simulator.app --args -CurrentDeviceUDID DE1F3042-4033-4A69-B0BF-
FD71713CFBF6
```

Make sure the simulator is launched and is sitting on the home screen. You can get to the home screen by pressing **Command** + **Shift** + **H**. Once your simulator is set up, head over to the Terminal window and attach LLDB to the SpringBoard application:

```
lldb -n SpringBoard
```

This attaches LLDB to the SpringBoard instance running on the iOS Simulator! SpringBoard is the program that controls the home screen on iOS.

Once attached, type the following into LLDB:

```
(lldb) p/x @"Yay! Debugging"
```

You should get some output similar to the following:

```
(__NSCFString *) $3 = 0x0000618000644080 @"Yay! Debugging!"
```

Take a note of the memory reference of this newly created `NSString` instance as you'll use it soon. Now, create a breakpoint on `UILabel`'s `setText:` method in LLDB:

```
(lldb) br set -n  "-[UILabel setText:]" -C "po $rdx = 0x0000618000644080"
-G1
```

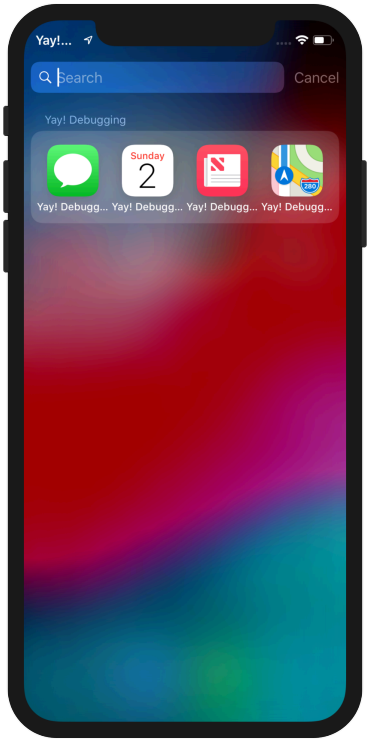
The above breakpoint will stop on the `-[UILabel setText:]` Objective-C method. When that happens, it will assign the RDX register the value `0x0000618000644080`, thanks to the `-C` or `--command` option. In addition, you've told LLDB to resume execution immediately after executing this command via the `-G` or `--auto-continue` option, which expects a boolean to determine if it should auto continue.

Take a step back and review what you've just done. Whenever `UILabel`'s `setText:` method gets hit, you're replacing what's in `RDX` — the third parameter — with a different `NSString` instance that says **Yay! Debugging!**.

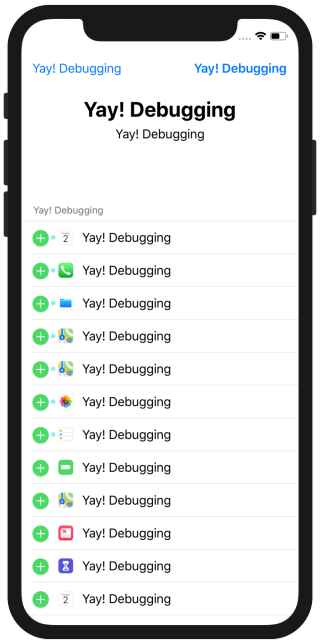
Resume the debugger by using the **`continue`** command:

```
(lldb) continue
```

Explore the SpringBoard Simulator app and see what content has changed. Swipe up and down and observe the changes:



Try exploring other areas where modal presentations can occur, as this will likely result in a new `UIViewController` (and all of its subviews) being lazily loaded, causing the breakpoint action to be hit.



Although this might seem like a cool gimmicky programming trick, it provides an insightful look into how a limited knowledge of registers and assembly can produce big changes in applications you don't have the source for.

This is also useful from a debugging standpoint, as you can quickly visually verify where the `-[UILabel setText:]` is executed within the SpringBoard application and run breakpoint conditions to find the exact line of code that sets a particular `UILabel`'s text.

To continue this thought, any `UILabel` instances whose text did not change also tells you something. For example, the `UIButtons` whose text didn't change to `Yay! Debugging!` speaks for itself. Perhaps the `UILabel`'s `setText:` was called at an earlier time? Or maybe the developers of the SpringBoard application chose to use `setAttributedText:` instead? Or maybe they're using a private method that is not publicly available to third-party developers?

As you can see, using and manipulating registers can give you a lot of insight into how an application functions.

Where to go from here?

Whew! That was a long one, wasn't it? Sit back and take a break with your favorite form of liquid; you've earned it.

So what did you learn?

- Architectures define a calling convention which dictates where parameters to a function and its return value are stored.
- In Objective-C, the `RDI` register is the reference of the calling `NSObject`, `RSI` is the Selector, `RDX` is the first parameter and so on.
- Even in Swift 4.2, there's still not a consistent register calling convention. For right now, the reference to "self" in a class is passed on the stack allowing the parameters to start with the `RDI` register. But who knows how long this will last and what crazy changes will take place before the Swift ABI will stabilize.
- The `RAX` register is used for return values in functions regardless of whether you're working with Objective-C or Swift.
- Make sure you use the Objective-C context when printing registers with `$`.

There's a lot you can do with registers. Try exploring apps you don't have the source code for; it's a lot of fun and will build a good foundation for tackling tough debugging problems.

Try attaching to an application on the iOS Simulator and map out the `UIViewControllers` as they appear using assembly, a smart breakpoint, and a breakpoint command.

Chapter 12: Assembly & Memory

You've begun the journey and learned the dark arts of the x64 calling convention in the previous chapter. When a function is called, you now know how parameters are passed to functions, and how function return values come back. What you haven't learned yet is how code is executed when it's loaded into memory.

In this chapter, you'll explore how a program executes. You'll look at a special register used to tell the processor where it should read the next instruction from, as well as how different sizes and groupings of memory can produce *very* different results.

Setting up the Intel-Flavored Assembly Experience™

As mentioned in the previous chapter, there are two main ways to display assembly. One type, **AT&T** assembly, is the default assembly set for LLDB. This flavor has the following format:

```
opcode  source  destination
```

Take a look at a concrete example:

```
movq  $0x78, %rax
```

This will move the hexadecimal value `0x78` into the `RAX` register. Although this assembly flavor is nice for some, you'll use the **Intel** flavor instead from here on out.

Why opt for Intel over AT&T? The answer can be best explained by this simple tweet...

> **Note**: In all seriousness, the choice of assembly flavor is somewhat of a flame war — check out this discussion in StackOverflow: https://stackoverflow.com/questions/972602/att-vs-intel-syntax-and-limitations.
>
> Using Intel was based on the admittedly loose consensus that Intel is better for reading, but at times, worse for writing. Since you're learning about debugging, the majority of time you'll be reading assembly as opposed to writing it.

Add the following lines to the bottom of your **~/.lldbinit** file:

```
settings set target.x86-disassembly-flavor intel
settings set target.skip-prologue false
```

The first line tells LLDB to display x86 assembly (both 32-bit and 64-bit) in the Intel flavor.

The second line tells LLDB to not skip the function prologue. You came across this earlier in this book, and from now on it's prudent to not skip the prologue since you'll be inspecting assembly right from the first instruction in a function.

> **Note**: When editing your **~/.lldbinit** file, make sure you don't use a program like TextEdit for this, as it will add unnecessary characters into the file that could result in LLDB not correctly parsing the file. An easy (although dangerous) way to add this is through a Terminal command like so: `echo "settings set target.x86-disassembly-flavor intel" >> ~/.lldbinit`.
>
> Make sure you have two '>>' in there or else you'll overwrite all your previous content in your **~/.lldbinit** file. If you're not comfortable with the Terminal, editors like `nano` (which you've used earlier) are your best bet.

The Intel flavor will swap the source and destination values, remove the '`%`' and '`$`' characters as well as do many, many other changes. Since you're not using the AT&T syntax, it's better to not explain the full differences between the two assembly flavors, and instead just learn the Intel format.

Take a look at the previous example, now shown in the Intel flavor and see how much cleaner it looks:

```
mov  rax, 0x78
```

Again, this will move the hexadecimal value `0x78` into the `RAX` register.

Compared to the AT&T flavor shown earlier, the Intel flavor swaps the source and destination operands. The destination operand now precedes the source operand. When working with assembly, it's important that you always identify the correct flavor, since a different action could occur if you're not clear which flavor you're working with.

From here on out, the Intel flavor will be the path forward. If you ever see a numeric hexadecimal constant that begins with a `$` character, or a register that begins with `%`, know that you're in the wrong assembly flavor and should change it using the process described above.

Creating the cpx command

First of all, you're going to create your own LLDB command to help later on.

Open `~/.lldbinit` again in your favorite text editor (`vim`, right?). Then add the following to the bottom of the file:

```
command alias -H "Print value in ObjC context in hexadecimal" -h "Print
in hex" -- cpx expression -f x -l objc --
```

This command, **`cpx`**, is a convenience command you can use to print out something in hexadecimal format, using the Objective-C context. This will be useful when printing out register contents.

Remember, registers aren't available in the Swift context, so you need to use the Objective-C context instead.

Now you have the tools needed to explore memory in this chapter through an assembly point of view!

Bits, bytes, and other terminology

Before you begin exploring memory, you need to be aware of some vocabulary about how memory is grouped.

A value that can contain either a 1 or a 0 is known as a **bit**. You can say there are 64 bits per address in a 64-bit architecture. Simple enough.

When there are 8 bits grouped together, they're known as a **byte**. How many unique values can a byte hold? You can determine that by calculating 2^8 which will be 256 values, starting from 0 and going to 255.

Lots of information is expressed in bytes. For example, the C `sizeof()` function returns the size of the object in bytes.

If you are familiar with ASCII character encoding, you'll recall all ASCII characters can be held in a single byte.

It's time to take a look at this terminology in action and learn some tricks along the way.

Open up the **Registers** macOS application, which you'll find in the resources folder for this chapter. Next, build and run the app. Once it's running, pause the program and bring up the LLDB console. As mentioned previously, this will result in the non-Swift debugging context being used.

```
(lldb) p sizeof('A')
```

This will print out the number of bytes required to make up the `A` character:

```
(unsigned long) $0 = 1
```

Next, type the following:

```
(lldb) p/t 'A'
```

You'll get the following output:

```
(char) $1 = 0b01000001
```

This is the binary representation for the character `A` in ASCII.

Another more common way to display a byte of information is using hexadecimal values. Two hexadecimal digits are required to represent a byte of information in hexadecimal.

Print out the hexadecimal representation of `A`:

```
(lldb) p/x 'A'
```

You'll get the following output:

```
(char) $2 = 0x41
```

Hexadecimal is great for viewing memory because a single hexadecimal digit represents exactly 4 bits. So if you have 2 hexadecimal digits, you have 1 byte. If you have 8 hexadecimal digits, you have 4 bytes. And so on.

Here are a few more terms for you that you'll find useful in the chapters to come:

- **Nybble**: 4 bits, a single value in hexadecimal
- **Half word**: 16 bits, or 2 bytes
- **Word**: 32 bits, or 4 bytes
- **Double word** or **Giant word**: 64 bits or 8 bytes.

With this terminology, you're all set to explore the different memory chunks.

The RIP register

Ah, the exact register to put on your gravestone.

When a program executes, code to be executed is loaded into memory. The location of which code to execute next in the program is determined by one magically important register: the **`RIP`** or **instruction pointer** register.

You'll now take a look at this register in action. Open the **Registers** application again and navigate to the **AppDelegate.swift** file. Modify the file so it contains the following code:

```
@NSApplicationMain
class AppDelegate: NSObject, NSApplicationDelegate {

  func applicationWillBecomeActive(
    _ notification: Notification) {
      print("\(#function)")
      self.aBadMethod()
  }

  func aBadMethod() {
    print("\(#function)")
  }

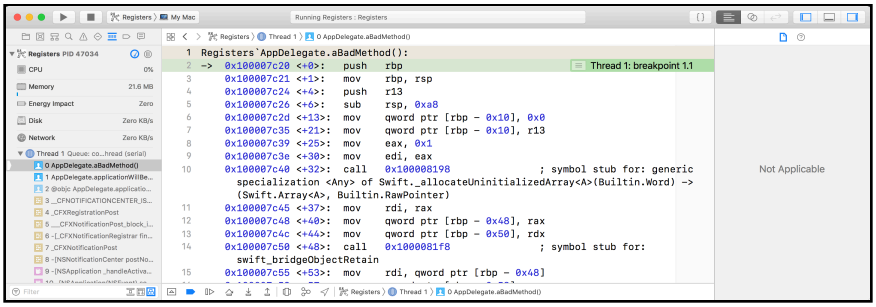
  func aGoodMethod() {
    print("\(#function)")
  }
}
```

Build and run the application. Unsuprisingly, the method name will get spat out in `applicationWillBecomeActive(_:)` to the debug console, followed by the `aBadMethod` output. There will be no execution of `aGoodMethod`.

Create a breakpoint at the very begining of the `aBadMethod` using the Xcode GUI:

```
 9 import Cocoa
10
11 @NSApplicationMain
12 class AppDelegate: NSObject, NSApplicationDelegate {
13
14   func applicationWillFinishLaunching(_ notification: Notification) {
15     print("\(#function)")
16     self.aBadMethod()
17   }
18
19   func aBadMethod() {
20     print("\(#function)")
21   }
22
23   func aGoodMethod() {
24     print("\(#function)")
25   }
26 }
27
28
```

Build and run again. Once the breakpoint is hit at the beginning of the `aBadMethod`, navigate to **Debug ▸ Debug Workflow ▸ Always Show Disassembly** in Xcode. You'll now see the actual assembly of the program!



Next, type the following into the LLDB console:

```
(lldb) cpx $rip
```

This prints out the instruction pointer register using the `cpx` command you created earlier.

You'll notice the output LLDB spits out will match the address highlighted by the green line in Xcode:

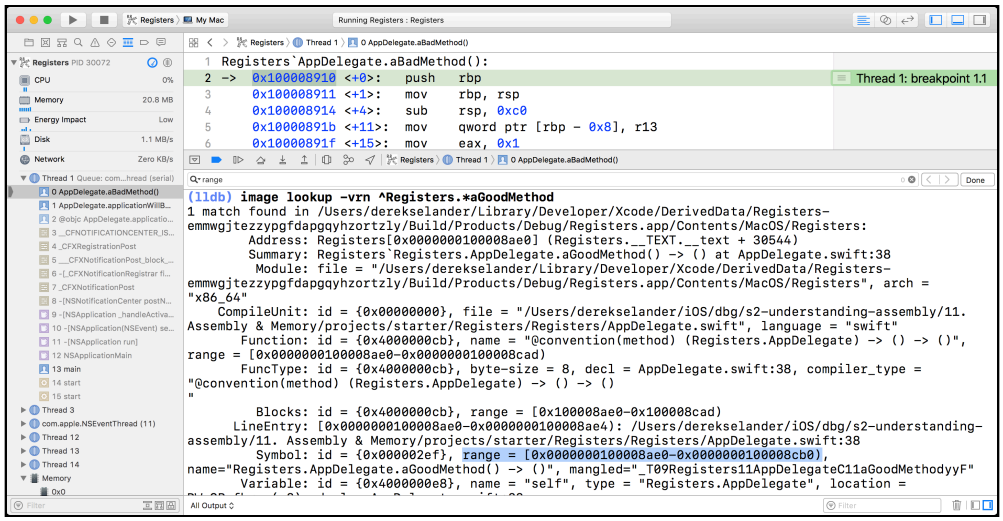
```
(unsigned long) $1 = 0x0000000100007c20
```

It's worth noting your address could be different than the above output, but the address of the green line and the `RIP` console output will match. Now, enter the following command in LLDB:

```
(lldb) image lookup -vrn ^Registers.*aGoodMethod
```

This is the tried-and-true `image lookup` command with the typical regular expression arguments plus an added argument, `–v`, which dumps the verbose output.

You'll get a fair bit of content. Search for the content immediately following `range = [`; **Command + F** will prove useful here. It's the first value in the range brackets that you're looking for.



This address is known as the **load address**. This is the actual physical address of this function in memory.

This differs from the usual output you've seen in the `image lookup` command, in that it only displays the offset of the function relative to the executable, also known as the **implementation offset**. When hunting for a function's address, it's important to differentiate the load address from the implementation offset in an executable, as it will differ.

Copy this new address at the beginning of the range brackets. For this particular example, the load address of `aGoodMethod` is located at `0x0000000100003a10`. Now, write this address which points the beginning of the `aGoodMethod` method to the `RIP` register.

```
(lldb) register write rip 0x0000000100003a10
```

Click **continue** using the Xcode debug button. It's important you do this instead of typing `continue` in LLDB, as there is a bug that will trip you up when modifying the `RIP` register and continuing in the console.

After pressing the Xcode continue button, you'll see that `aBadMethod()` is not executed and `aGoodMethod()` is executed instead. Verify this by viewing the output in the console log.

Note: Modifying the `RIP` register is actually a bit dangerous. You need to make sure the registers holding data for a previous value in the `RIP` register do not get applied to a new function which would make an incorrect assumption with the registers. Since `aGoodMethod` and `aBadMethod` are very similar in functionality, you've stopped at the beginning, and as no optimizations were applied to the Registers application, this is not a worry.

Registers and breaking up the bits

As mentioned in the previous chapter, x64 has 16 general purpose registers: `RDI`, `RSI`, `RAX`, `RDX`, `RBP`, `RSP`, `RCX`, `RDX`, `R8`, `R9`, `R10`, `R11`, `R12`, `R13`, `R14` and `R15`.

In order to maintain compatibility with previous architectures, such as i386's 32-bit architecture, registers can be broken up into their 32, 16, or 8-bit values.

For registers that have had a history across different architectures, the frontmost character in the name given to the register determines the size of the register. For example, the `RIP` register starts with `R`, which signifies 64 bits. If you wanted the 32 bit equivalent of the RIP register, you'd swap out the `R` character with an `E`, to get the `EIP` register.

64-bit Register	32-bit Register	16-bit Register	8-bit Register
rax	eax	ax	al
rbx	ebx	bx	bl
rcx	ecx	cx	cl
rdx	edx	dx	dl
rsi	esi	si	sil
rdi	edi	di	dil
rbp	ebp	bp	bpl
rsp	esp	sp	spl
r8	r8d	r8w	r8l
r9	r9d	r9w	r9l
r10	r10d	r10w	r10l
r11	r11d	r11w	r11l
r12	r12d	r12w	r12l
r13	r13d	r13w	r13l
r14	r14d	r14w	r14l
r15	r15d	r15w	r15l

Why is this useful? When working with registers, sometimes the value passed into a register does not need to use all 64 bits. For example, consider the Boolean data type.

All you really need is a 1 or a 0 to indicate `true` or `false`, right? Based upon the languages features and constraints, the compiler knows this and will sometimes only write information to certain parts of a register.

Let's see this in action.

Remove all breakpoints in the Registers project. Build and run the project. Now, pause the program out of the blue.

Once stopped, type the following:

```
(lldb) register write rdx 0x0123456789ABCDEF
```

This writes a value to the `RDX` register.

Let's halt for a minute. A word of warning: You should be aware that writing to registers could cause your program to tank, especially if the register you write to is expected to have a certain type of data. But you're doing this in the name of science, so don't worry if your program does crash!

Confirm that this value has been successfully written to the `RDX` register:

```
(lldb) p/x $rdx
```

Since this is a 64-bit program, you'll get a double word, i.e. 64 bits, or 8 bytes, or 16 hexadecimal digits.

Now, try printing out the `EDX` register:

```
(lldb) p/x $edx
```

The `EDX` register is the least-significant half of the `RDX` register. So you'll only see the least-significant half of the double word, i.e., a word. You should see the following:

```
0x89abcdef
```

Next, type the following:

```
(lldb) p/x $dx
```

This will print out the `DX` register, which is the least-significant half of the `EDX` register. It is therefore a half word. You should see the following:

```
0xcdef
```

Next, type the following:

```
(lldb) p/x $dl
```

This prints out the DL register, which is the least-significant half of the DX register — a byte this time. You should see the following:

```
0xef
```

Finally, type the following:

```
(lldb) p/x $dh
```

This gives you the most significant half of the DX register, i.e. the other half to that given by DL. It should come as no surprise that the L in DL stands for "low" and the H in DH stands for "high".

Keep an eye out for registers with different sizes when exploring assembly. The size of the registers can give clues about the values contained within. For example, you can easily hunt down functions that return Booleans by looking for registers having the L suffix, since a Boolean needs only a single bit to be used.

Registers R8 to R15

Since the R8 to R15 family of registers were created only for 64-bit architectures, they use a completely different format for signifying their smaller counterparts.

Now you'll explore R9's different sizing options. Build and run the Registers application, and pause the debugger. Like before, write the same hex value to the R9 register:

```
(lldb) register write $r9 0x0123456789abcdef
```

Confirm that you've set the R9 register by typing the following:

```
(lldb) p/x $r9
```

Next type the following:

```
(lldb) p/x $r9d
```

This will print the lower 32 bits of the R9 register. Note how it's different than how you specified the lower 32 bits for RDX (that is, EDX, if you've forgotten already).

Next, type the following:

```
(lldb) p/x $r9w
```

This time you get the lower 16 bits of R9. Again, this is different than how you did this for RDX.

Finally, type the following:

```
(lldb) p/x $r9l
```

This prints out the lower 8 bits of `R9`.

Although this seems a bit tedious, you're building up the skills to read an onslaught of assembly.

Breaking down the memory

Now that you've taken a look at the instruction pointer, it's time to further explore the memory behind it. As its name suggests, the instruction pointer is actually a **pointer**. It's not executing the instructions stored in the `RIP` register — it's executing the instructions pointed to in the RIP register.

Seeing this in LLDB will perhaps describe it better. Back in the Registers application, open **AppDelegate.swift** and once again set a breakpoint on `aBadMethod`. Build and run the app.

Once the breakpoint is hit and the program is stopped, navigate back to the assembly view. If you forgot, and haven't created a keyboard shortcut for it, it's found under **Debug ▸ Debug Workflow ▸ Always Show Disassembly**.

You'll be greeted by the onslaught of opcodes and registers. Take a look at the location of the `RIP` register, which should be pointing to the very beginning of the function.

For this particular build, the beginning address of `aBadMethod` begins as **`0x100007c20`**. As usual, your address will likely be different.

In the LLDB console, type the following:

```
(lldb) cpx $rip
```

As you know by now, this prints out the contents of the instruction pointer register.

As expected, you'll get the address of the start of `aBadMethod`. But again, the `RIP` register points to a value in memory. What is it pointing to? Well… you could dust off your mad C coding skillz (you remember those, right?) and dereference the pointer, but there's a much more elegant way to go about it using LLDB.

Type the following, replacing the address with the address of your `aBadMethod` function:

```
(lldb) memory read -fi -c1 0x100007c20
```

Wow, what the heck does that command do?!

`memory read` takes a value and reads the contents pointed at by the memory address you supply. The `-f` command is a formatting argument; in this case, it's the assembly **instruction** format. Finally you're saying you only want one assembly instruction to be printed out with the **count**, or `-c` argument.

You'll get output that looks similar to this:

```
  ->  0x100007c20:  55  push   rbp
```

This here is some gooooooooood output. It's telling you the assembly instruction, as well as the opcode, provided in hexadecimal (**0x55**) that is responsible for the `pushq rbp` operation.

Look at that `"55"` there in the output some more. This is an encoding of the entire instruction, i.e. the whole `pushq rbp`. Don't believe me? You can verify it. Type the following into LLDB:

```
  (lldb) expression -f i -l objc -- 0x55
```

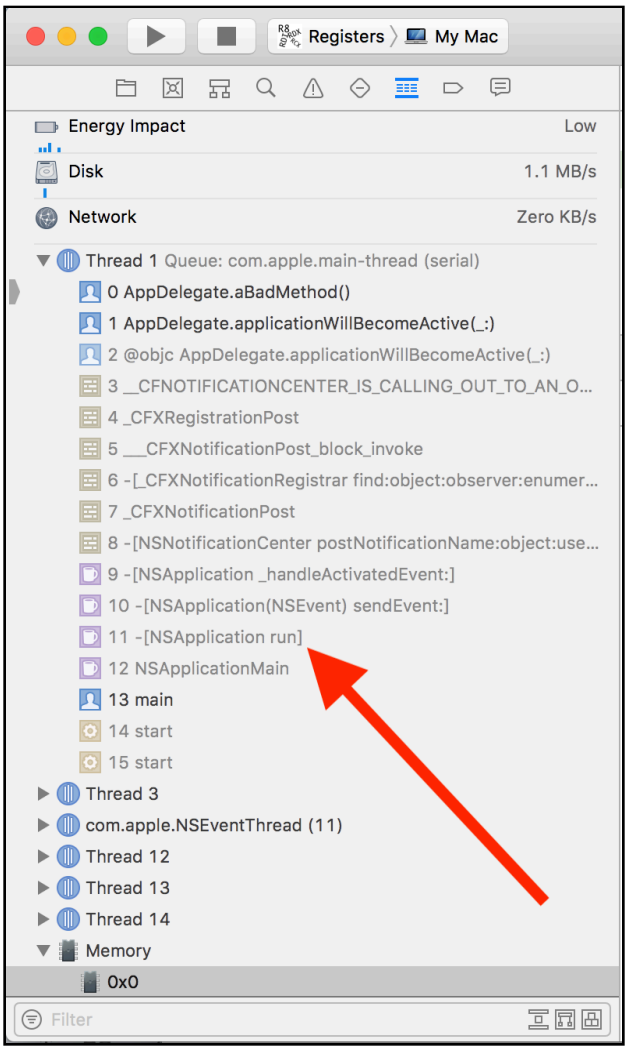
This effectively asks LLDB to decode `0x55`. You'll get the following output:

```
  (int) $0 = 55  push   rbp
```

That command is a little long, but it's because you need the required switch to Objective-C context if you are in the Swift debugging context. However, if you move to the Objective-C debugging context, you can use a convenience expression that is a lot shorter.

Try clicking on a different frame in the left panel of Xcode to get into an Objective-C context which doesn't contain Swift or Objective-C/Swift bridging code.

Click on any frame which is in an Objective-C function.



Next, type the following into the LLDB console:

```
(lldb) p/i 0x55
```

Much better, right?

Now, back to the application in hand. Type the following into LLDB, replacing the address once again with your `aBadMethod` function address:

```
(lldb) memory read -fi -c4 0x100007c20
```

You'll get 10x the output! That's something worthy to put on that LinkedIn résumé…

By the way, there's a shorthand convenience way to execute the above command. You can simply type the following to achieve the same result.

```
(lldb) x/4i 0x100007c20
```

With either command you choose, you'll get something similar to following output:

```
0x100007c20: 55                    push rbp
0x100007c21: 48 89 e5              mov  rbp, rsp
0x100007c24: 41 55                 push r13
0x100007c26: 48 81 ec a8 00 00 00  sub  rsp, 0xa8
```

There's something interesting to note here: assembly instructions can have variable lengths. Take a look at the first instruction, versus the rest of the instructions in the output. The first instruction is 1 byte long, represented by `0x55`. The following instruction is 3 bytes long.

Make sure you are still in an Objective-C context, and try to print out the opcode responsible for this instruction. It's just 3 bytes, so all you have to do is join them together, right?

```
(lldb) p/i 0x4889e5
```

You'll get a different instruction completely unrelated to the `mov %rsp, %rbp` instruction! You'll see this:

```
e5 89  inl    $0x89, %eax
```

What gives? Perhaps now would be a good time to talk about **endianness**.

Endianness... this stuff is reversed?

The x64 as well as the ARM family architecture devices all use **little-endian**, which means that data is stored in memory with the least significant byte first. If you were to store the number `0xabcd` in memory, the `0xcd` byte would be stored first, followed by the `0xab` byte.

Back to the instruction example, this means that the instruction `0x4889e5` will be stored in memory as `0xe5`, followed by `0x89`, followed by `0x48`.

Jumping back to that `mov` instruction you encountered earlier, try reversing the bytes that used to make up the assembly instruction. Type the following into LLDB:

```
(lldb) p/i 0xe58948
```

You'll now get your expected assembly instruction:

```
(Int) $R1 = 48 89 e5  mov    rbp, rsp
```

Let's see some more examples of little-endian in action. Type the following into LLDB:

```
(lldb) memory read -s1 -c20 -fx 0x100003840
```

This command reads the memory at address `0x100003840`. It reads in size chunks of 1 byte thanks to the `-s1` option, and a count of 20 thanks to the `-c20` option. You'll see something like this:

```
0x100003840: 0x55 0x48 0x89 0xe5 0x48 0x83 0xec 0x60
0x100003848: 0xb8 0x01 0x00 0x00 0x00 0x89 0xc1 0x48
0x100003850: 0x89 0x7d 0xf8 0x48
```

Now, double the size and half the count like so:

```
(lldb) memory read -s2 -c10 -fx 0x100003840
```

You will see something like this:

```
0x100003840: 0x4855 0xe589 0x8348 0x60ec 0x01b8 0x0000 0x8900 0x48c1
0x100003850: 0x7d89 0x48f8
```

Notice how when the memory values are grouped together, they are reversed thanks to being in little-endian.

Now double the size and half the count again:

```
(lldb) memory read -s4 -c5 -fx 0x100003840
```

And now you'll get something like this:

```
0x100003840: 0xe5894855 0x60ec8348 0x000001b8 0x48c18900
0x100003850: 0x48f87d89
```

Once again the values are reversed compared to the previous output.

This is *very* important to remember and also a source of confusion when exploring memory. Not only will the size of memory give you a potentially incorrect answer, but also the order. Remember this when you start yelling at your computer when you're trying to figure out how something should work!

Where to go from here?

Good job getting through this one. Memory layout can be a confusing topic. Try exploring memory on other devices to make sure you have a solid understanding of the little-endian architecture and how assembly is grouped together.

In the next chapter, you'll explore the stack frame and how a function gets called.

Chapter 13: Assembly & the Stack

In x86_64, when there are more than six parameters passed into a function, the excess parameters are passed through the stack (there's situations when this is not true, but one thing at a time, young grasshopper). But what does *being passed on the stack* mean exactly? It's time to take a deeper dive into what happens when a function is called from an assembly standpoint by exploring some "stack related" registers as well as the contents in the stack.

Understanding how the stack works is useful when you're reverse engineering programs, since you can help deduce what parameters are being manipulated in a certain function when no debugging symbols are available.

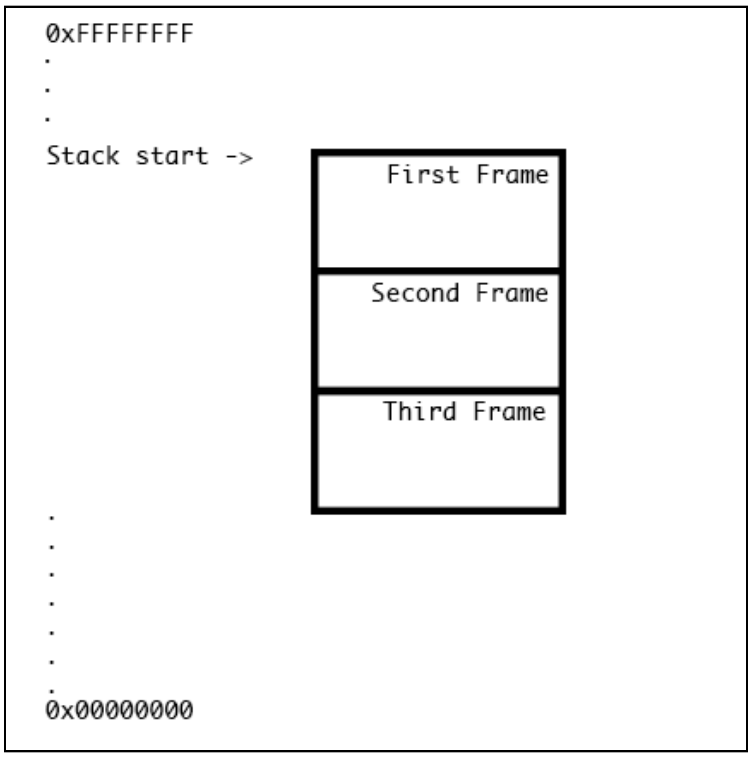
Let's begin.

The stack, revisited

As discussed previously in Chapter 6, "Thread, Frame & Stepping Around", when a program executes, the memory is laid out so the stack starts at a "high address" and grows downward, towards a lower address; that is, towards the heap.

> **Note**: In some architectures, the stack grows upwards. But for x64 and ARM for iOS devices, the two you care about, both grow the the stack downwards.

Confused? Here's an image to help clarify how the stack moves.



The stack starts at a high address. How high, exactly, is determined by the operating system's kernel. The kernel gives stack space to each running program (well, each thread).

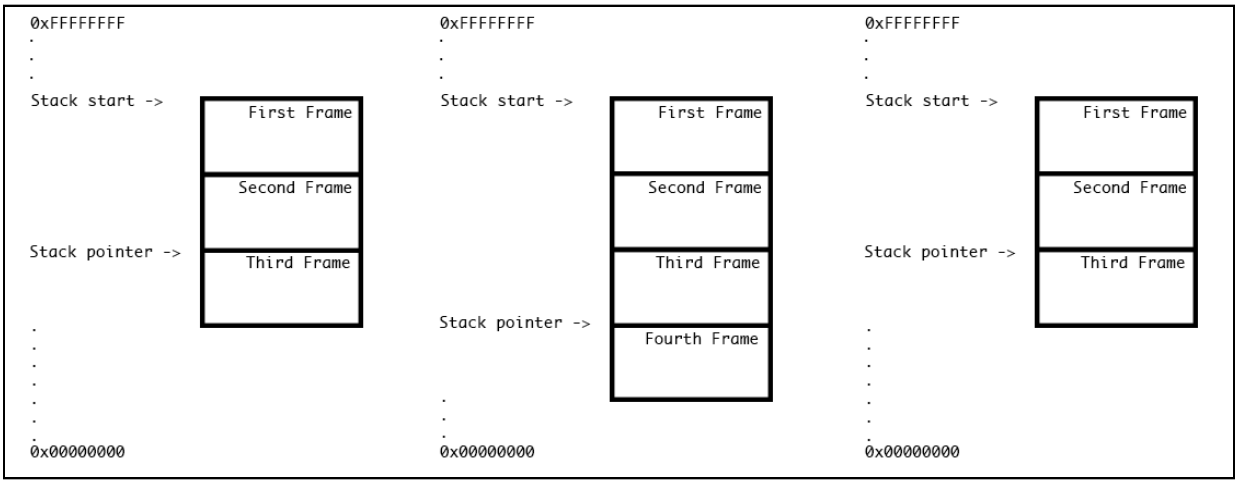
The stack is finite in size and increases by growing downwards in memory address space. As space on the stack is used up, the pointer to the "top" of the stack moves down from the highest address to the lowest address.

Once the stack reaches the finite size given by the kernel, or if it crosses the bounds of the heap, the stack is said to *overflow*. This is a fatal error, often referred to as a *stack overflow*. Now you know where your favorite website gets its name from!

Stack pointer & base pointer registers

Two very important registers you've yet to learn about are the **RSP** and **RBP**. The stack pointer register, RSP, points to the head of the stack for a particular thread. The head of the stack will grow downwards, so the RSP will decrement when items are added to the stack. The RSP will *always* point to the head of the stack.

Here's a visual of the stack pointer changing when a function is called.



In the above image, the sequence of the stack pointer follows:

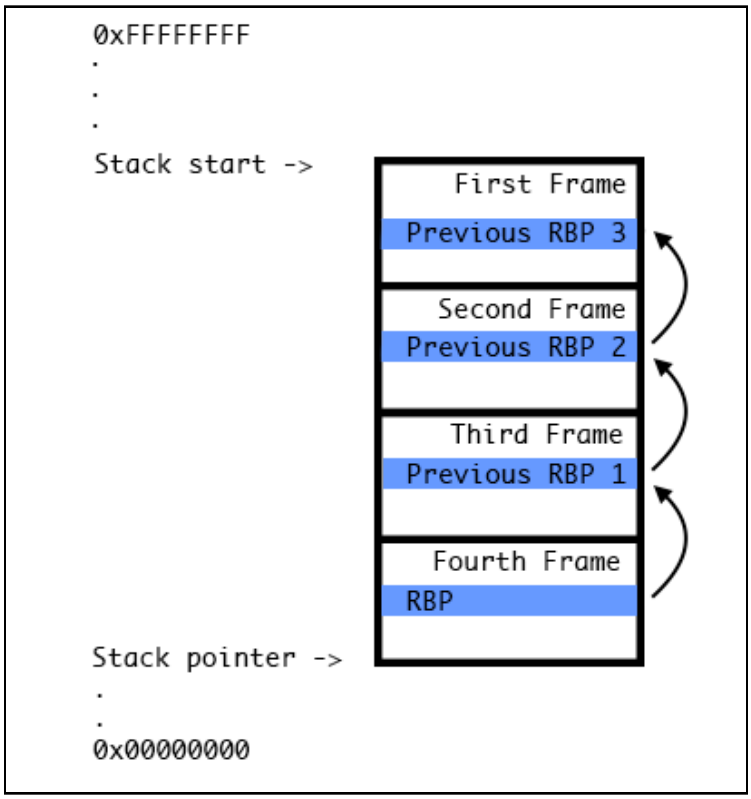
1. The stack pointer currently points to Frame 3.
2. The code pointed to by the instruction pointer register calls a new function. The stack pointer gets updated to point to a new frame, Frame 4, which is potentially responsible for scratchspace and data inside this newly called function from the instruction pointer.
3. Execution is completed in Frame 4 and control resumes back in Frame 3. The stack pointer's previous reference to Frame 4 gets popped off and resumes pointing to Frame 3

The other important register, the base pointer register (`RBP`), has multiple uses during a function being executed. Programs use offsets from the `RBP` to access local variables or function parameters while execution is inside the method/function. This happens because the `RBP` is set to the value of the `RSP` register at the beginning of a function in the **function prologue**.

The interesting thing here is the previous contents of the base pointer are stored on the stack *before* it's set to the value of the `RSP` register. This is the first thing that happens in the function prologue. Since the base pointer is saved onto the stack and set to the current stack pointer, you can traverse the stack just by knowing the value in the base pointer register. A debugger does this when it shows you the stack trace.

Note: Some systems don't use a base pointer, and it's possible to compile your application to omit using the base pointer. The logic is it might be beneficial to have an extra register to use. But this means you can't unwind the stack easily, which makes debugging much harder.

Yeah, an image is definitely needed to help explain.



When a function prologue is finished setting up, the contents of `RBP` will point to the previous `RBP` a stack frame lower.

Note: When you jump to a different stack frame by clicking on a frame in Xcode or using LLDB, both the `RBP` & `RSP` registers will change values to correspond to the new frame! This is expected because local variables for a function use offsets of `RBP` to get their values.

If the `RBP` didn't change, you'd be unable to print local variables to that function, and the program might even crash. This might result in a source of confusion when exploring the `RBP` & `RSP` registers, so always keep this in mind. You can verify this in LLDB by selecting different frames and typing `cpx $rbp` or `cpx $rsp` in the LLDB console.

So why are these two registers important to learn about? When a program is compiled with debug information, the debug information references offsets from the base pointer register to get a variable. These offsets are given names, the same names you gave your variables in your source code.

When a program is compiled and optimized for release, the debug information that comes packaged into the binary is removed. Although the names to the references of these variables and parameters are removed, you can still use offsets of the stack pointer and base pointer to find the location of where these references are stored.

Stack related opcodes

So far, you've learned about the calling convention and how the memory is laid out, but haven't really explored what the many opcodes actually *do* in x64 assembly. It's time to focus on several stack related opcodes in more detail.

The 'push' opcode

When anything such as an `int`, Objective-C instance, Swift class or a reference needs to be saved onto the stack, the **`push`** opcode is used. `push` decrements the stack pointer (remember, the stack grows downward), then stores the value assigned to the memory address pointed at by the new `RSP` value.

After a `push` instruction, the most recently pushed value will be located at the address pointed to by `RSP`. The previous value would be at `RSP` plus the size of the most recently pushed value — usually 8 bytes for 64-bit architecture.

To see at a concrete example, consider the following opcode:

```
push 0x5
```

This would decrement the `RSP`, then store the value `5` in the memory address pointed to by `RSP`. So, in C pseudocode:

```
RSP = RSP - 0x8
*RSP = 0x5
```

The 'pop' opcode

The **pop** opcode is the exact opposite of the `push` opcode. `pop` takes the value from the `RSP` register and stores it to a destination. Next, the `RSP` is incremented by `0x8` because, again, as the stack gets smaller, it will grow to a higher address.

Below is an example of `pop`:

```
pop rdx
```

This stores the value of the `RSP` register into the `RDX` register, then increments the `RSP` register. Here's the pseudocode below:

```
RDX = *RSP
RSP = RSP + 0x8
```

The 'call' opcode

The **`call`** opcode is responsible for executing a function. `call` pushes the address of where to return to after the called function completes; then jumps to the function.

Imagine a function at `0x7fffb34df410` in memory like so:

```
0x7fffb34de913 <+227>: call   0x7fffb34df410
0x7fffb34de918 <+232>: mov    edx, eax
```

When an instruction is executed, first the `RIP` register is incremented, then the instruction is executed. So, when the `call` instruction is executed, the `RIP` register will increment to `0x7fffb34de918`, then execute the instruction pointed to by `0x7fffb34de913`. Since this is a `call` instruction, the `RIP` register is pushed onto the stack (just as if a `push` had been executed) then the `RIP` register is set to the value `0x7fffb34df410`, the address of the function to be executed.

The pseudocode would look similar to the following:

```
RIP = 0x7fffb34de918
RSP = RSP - 0x8
*RSP = RIP
RIP = 0x7fffb34df410
```

From there, execution continues at the location `0x7fffb34df410`.

Computers are pretty cool, aren't they?

The 'ret' opcode

The **`ret`** opcode is the opposite of the `call` opcode, in that it pops the top value off the stack (which will be the return address pushed on by the `call` opcode, provided the assembly's pushes and pops match) then sets the `RIP` register to this address. Thus execution goes back to where the function was called from.

Now that you have a basic understanding of these four important opcodes, it's time to see them in action.

It's *very* important to have all `push` opcodes match your `pop` opcodes, or else the stack will get out of sync. For example, if there was no corresponding `pop` for a `push`, when the `ret` happened at the end of the function, the wrong value would be popped off. Execution would return to some random place, potentially not even a valid place in the program.

Fortunately, the compiler will take care of synchronizing your `push` and `pop` opcodes. You only need to worry about this when you're writing your own assembly.

Observing RBP & RSP in action

Now that you have an understanding of the `RBP` and `RSP` registers, as well as the four opcodes that manipulate the stack, it's time to see it all in action.

In the Registers application lives a function named **`StackWalkthrough(int)`**. This C function takes one integer as a parameter and is written in assembly (AT&T assembly, remember to be able to spot the correct location for the source and destination operands) and is located in **StackWalkthrough.s**. Open this file and have a look around; there's no need to understand it all just now. You'll learn how it works in a minute.

This function is made available to Swift through a bridging header **Registers-Bridging-Header.h**, so you can call this method written in assembly from Swift.

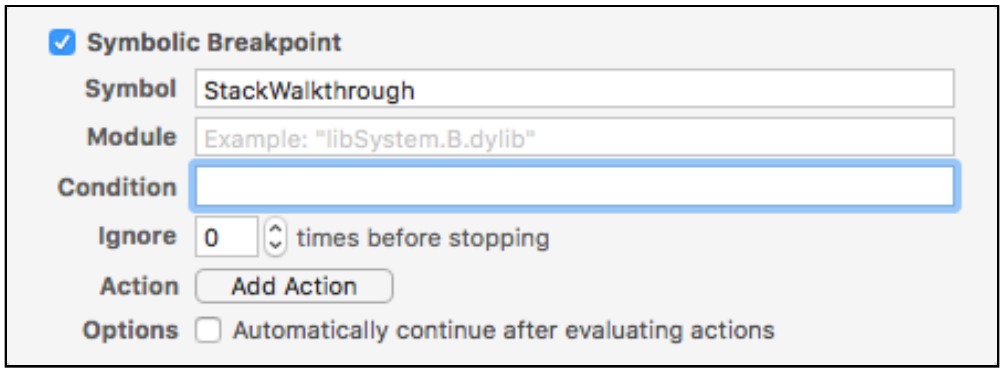
Now to make use of this.

Open **ViewController.swift**, and add the following below `viewDidLoad()`:

```
override func awakeFromNib() {
  super.awakeFromNib()
  StackWalkthrough(5)
}
```

This will call `StackWalkThrough` with a parameter of `5`. The `5` is simply a value used to show how the stack works.

Before exploring `RSP` and `RBP` in depth, it's best to get a quick overview of what is happening in `StackWalkthrough`. Create a **symbolic breakpoint** on the `StackWalkthrough` function.



Once created, build and run.

Xcode will break on `StackWalkthrough`. Be sure to view the StackWalkthrough function through "source" (even though it's assembly). Viewing the function through source will showcase the AT&T assembly (because it was written in AT&T ASM).

Xcode will display the following assembly:

```
push  %rbp       ; Push contents of RBP onto the stack (*RSP = RBP, RSP
decreases)

movq  %rsp, %rbp ; RBP = RSP
movq  $0x0, %rdx ; RDX = 0
movq  %rdi, %rdx ; RDX = RDI
push  %rdx       ; Push contents of RDX onto the stack (*RSP = RDX, RSP
decreases)

movq  $0x0, %rdx ; RDX = 0
pop   %rdx       ; Pop top of stack into RDX (RDX = *RSP, RSP increases)

pop   %rbp       ; Pop top of stack into RBP (RBP = *RSP, RSP increases)

ret              ; Return from function (RIP = *RSP, RSP increases)
```

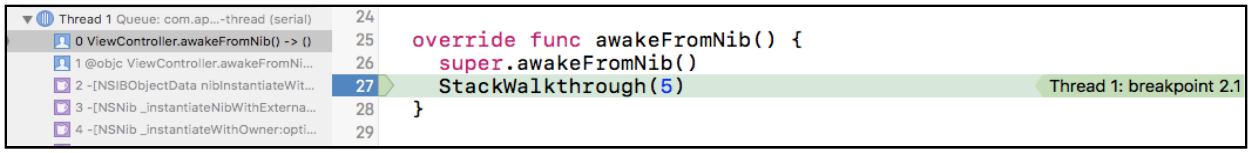
Comments have been added to help understand what's happening. Read it through and try to understand it if you can. You're already familiar with the `mov` instruction, and the rest of the assembly consists of function related opcodes you've just learned about.

This function takes the integer parameter passed into it (as you'll recall, the first parameter is passed in `RDI`), stores it into the `RDX` register, and pushes this parameter onto the stack. `RDX` is then set to `0x0`, then the value popped off the stack is stored back into the RDX register.

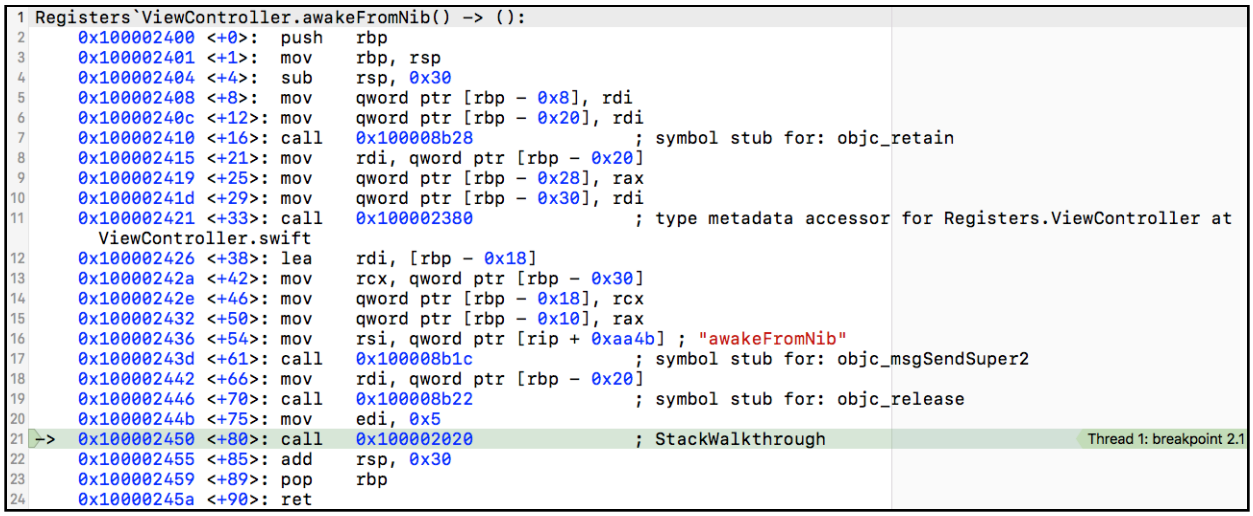
Make sure you have a good mental understanding of what is happening in this function, as you'll be exploring the registers in LLDB next.

Back in Xcode, create a breakpoint using Xcode's GUI on the `StackWalkthrough(5)` line in the `awakeFromNib` function of **ViewController.swift**. Leave the previous `StackWalkthrough` symbolic breakpoint alive, since you'll want to stop at the beginning of the `StackWalkthrough` function when exploring the registers.

Build and run and wait for the GUI breakpoint to trigger.



Now click **Debug ▸ Debug Workflow ▸Always Show Disassembly**, to show the disassembly. You'll be greeted with scary looking stuff!



Wow! Look at that! You've landed right on a `call` opcode instruction. Do you wonder what function you're about to enter?

> **Note**: If you didn't land right on the `call` instruction using the Xcode GUI breakpoint, you can either use LLDB's **thread step-inst** or more simply, **si** to single step through assembly instructions. Alternatively, you can create a GUI breakpoint on the memory address that `calls` the `StackWalkthrough` function.

From here on out, you'll step through every assembly instruction while printing out four registers of interest: `RBP`, `RSP`, `RDI` and `RDX`. To help with this, type the following into LLDB:

```
(lldb) command alias dumpreg register read rsp rbp rdi rdx
```

This creates the command **`dumpreg`** that will dump the four registers of interest. Execute `dumpreg` now:

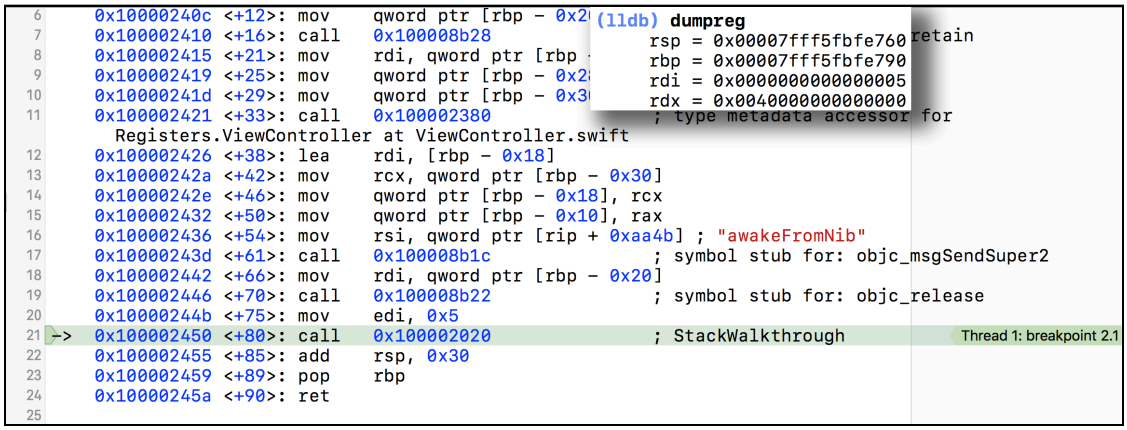
```
(lldb) dumpreg
```

You'll see something similar to the following:

```
rsp = 0x00007fff5fbfe820
rbp = 0x00007fff5fbfe850
rdi = 0x0000000000000005
rdx = 0x0040000000000000
```

For this section, the output of `dumpreg` will be overlaid on each assembly instruction to show exactly what is happening with each of the registers during each instruction. Again, even though the values are provided for you, it's very important you execute and understand these commands yourself.

Your screen will look similar to the following:



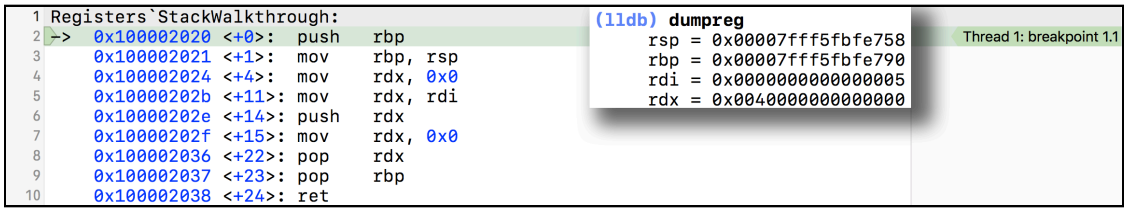
Once you jump into the function call, keep a *very* close eye on the `RSP` register, as it's about to change once `RIP` jumps to the beginning of `StackWalkthrough`. As you've learned earlier, the `RDI` register will contain the value for the first parameter, which is `0x5` in this case.

In LLDB, type the following:

```
(lldb) si
```

This is an alias for `thread step-inst`, which tells LLDB to execute the next instruction and then pause the debugger.

You've now stepped into `StackWalkthrough`. Again for each step, dump out the registers using `dumpreg`.



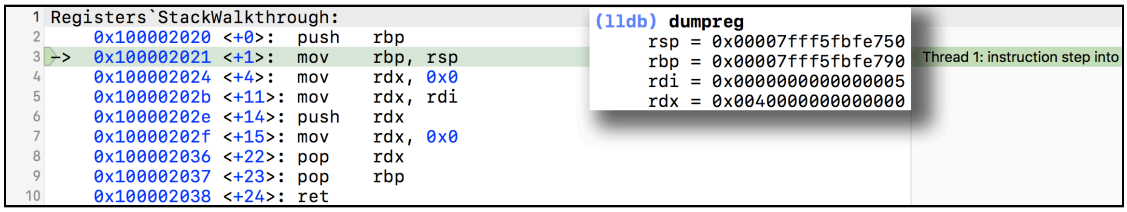
Take note of the difference in the `RSP` register. The value pointed at by `RSP` will now contain the return address to the previous function. For this particular example, `RSP`, which points to `0x7fff5fbfe758`, will contain the value `0x100002455` — the address immediately following the `call` in `awakeFromNib`.

Verify this now through LLDB:

```
(lldb) x/gx $rsp
```

The output will match the address immediately following the call opcode in `awakeFromNib`.

Next, perform an `si`, then `dumpreg` for the next instruction.



The value of `RBP` is pushed onto the stack. This means the following two commands will produce the same output. Execute both of them to verify.

```
(lldb) x/gx $rsp
```

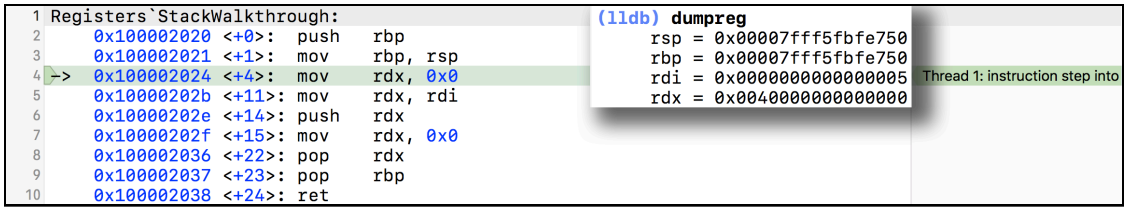
This looks at the memory address pointed at by the stack pointer register.

> **Note**: Wait, I just threw a new command at you with no context. The `x` command is a shortcut for the `memory read` command.
>
> The `/gx` says to format the memory in a giant word (8 bytes, remember that terminology from Chapter 12, "Assembly & Memory"?) in hexadecimal format.
>
> The weird formatting is due to the popularity of this command in `gdb`, which saw this command syntax ported into `lldb` to make the transition from debuggers easier.

Now look at the value in the base pointer register.

```
(lldb) p/x $rbp
```

Next, step into the next instruction, using `si` again:

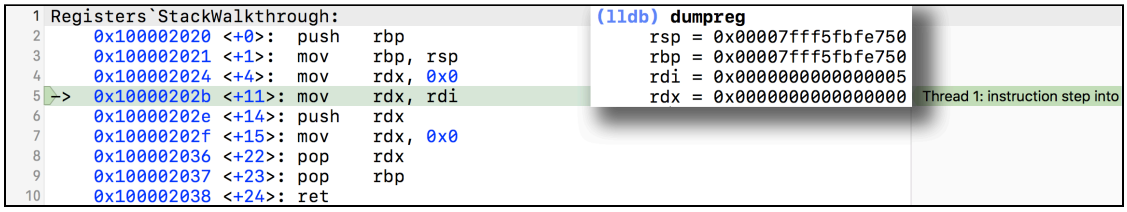


The base pointer is assigned to the value of the stack pointer. Verify both have the same value using `dumpreg` as well as the following LLDB command:

```
(lldb) p (BOOL)($rbp == $rsp)
```

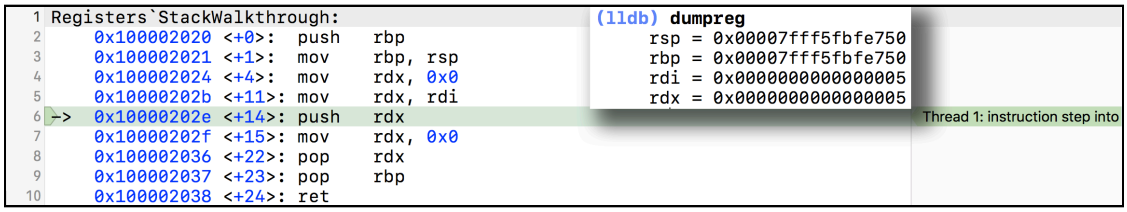
It's important you put parentheses around the expression, else LLDB won't parse it correctly.

Execute `si` and `dumpreg` again. This time it looks like the following:



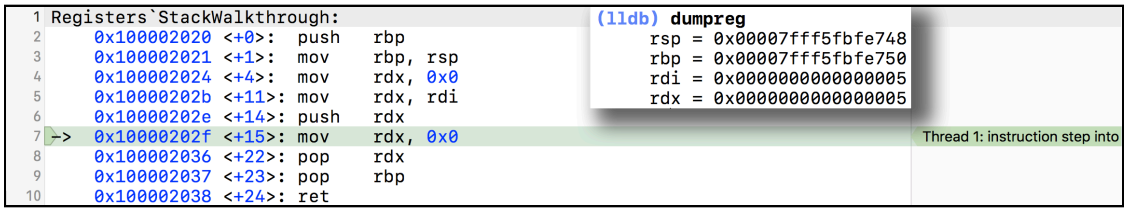
`RDX` is cleared to 0.

Execute `si` and `dumpreg` again. This time the output looks the following:



`RDX` is set to `RDI`. You can verify both have the same value with `dumpreg` again.

Execute `si` and `dumpreg`. This time it looks the following:



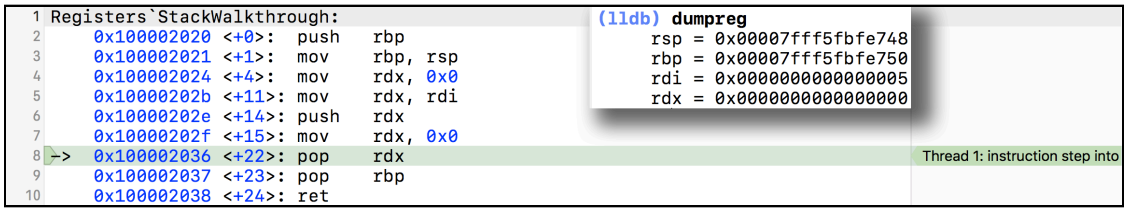
`RDX` is pushed onto the stack. This means the stack pointer was decremented, and `RSP` points to a value which will point to the value of `0x5`. Confirm that now:

```
(lldb) p/x $rsp
```

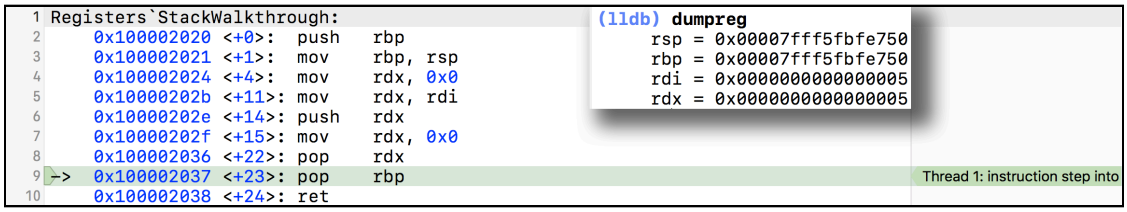
This gives the current value pointed at `RSP`. What does the value here point to?

```
(lldb) x/gx $rsp
```

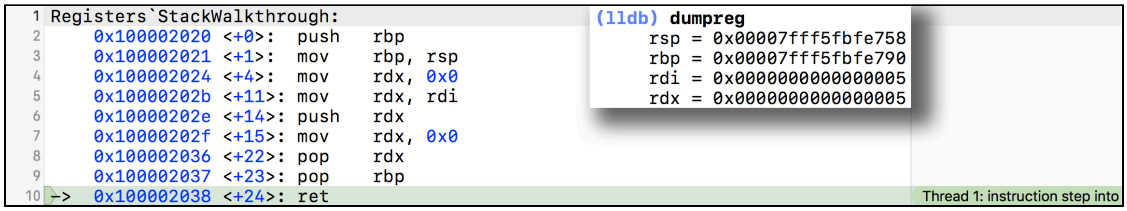
You'll get the expected `0x5`. Type `si` again to execute the next instruction:



`RDX` is set to `0x0`. Nothing too exciting here, move along... move along. Type `si` and `dumpreg` again:

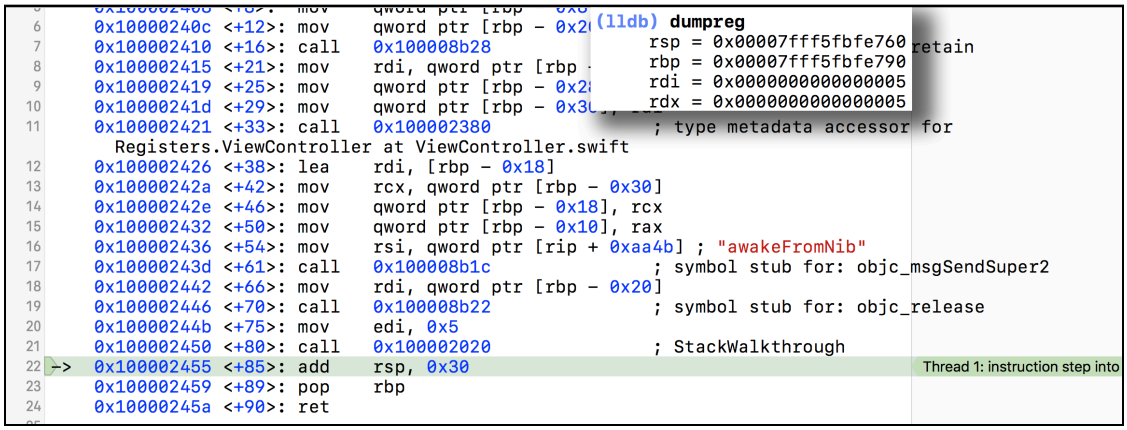


The top of the stack is popped into RDX, which you know was recently set to 0x5. The RSP is incremented by 0x8. Type si and dumpreg again:



The base pointer is popped off of the stack and reassigned back to the value it originally had when entering this function. The calling convention specifies RBP should remain consistent across function calls. That is, the RBP can't change to a different value once it leaves a function, so we're being a good citizen and restoring its value.

Onto the ret opcode. Keep an eye out for the RSP value about to change. Type si and dumpreg again:



The return address was pushed off the stack and set to the RIP register; you know this because you've gone back to where the function was called. Control then resumes in awakeFromNib,

Wowza! That was fun! A simple function, but it illustrates how the stack works through call, push, pop and ret instructions.

The stack and 7+ parameters

As described in Chapter 11, the calling convention for x86_64 will use the following registers for function parameters in order: RDI, RSI, RDX, RCX, R8, R9. When a function requires more than six parameters, the stack needs to be used.

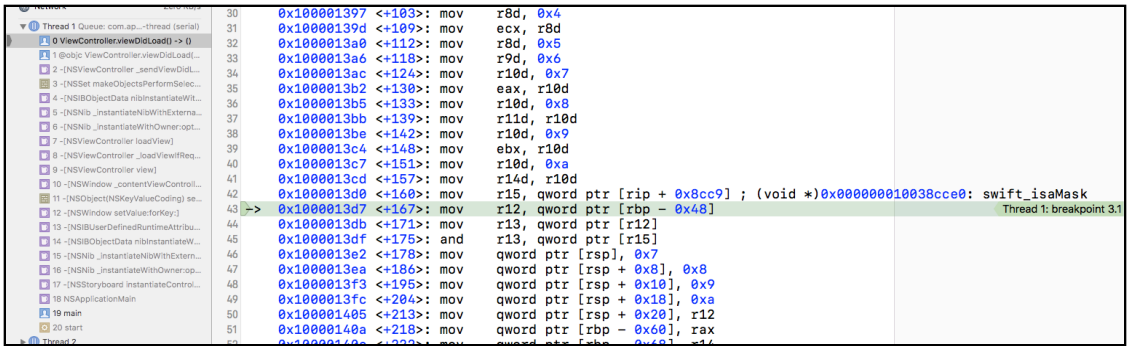
> **Note**: The stack may also need to be used when a large struct is passed to a function. Each parameter register can only hold 8 bytes (on 64-bit architecture), so if the struct needs more than 8 bytes, it will need to be passed on the stack as well. There are strict rules defining how this works in the calling convention, which all compilers must adhere to.

Open **ViewController.swift** and find the function named `executeLotsOfArguments(one:two:three:four:five:six:seven:eight:nine:ten:)`. You used this function in Chapter 11 to explore the registers. You'll use it again now to see how parameters 7 and beyond get passed to the function.

Add the following code to the end of `viewDidLoad`:

```
_ = self.executeLotsOfArguments(one: 1, two: 2, three: 3,
                                 four: 4, five: 5, six: 6,
                                 seven: 7, eight: 8, nine: 9,
                                 ten: 10)
```

Next, using the Xcode GUI, create a breakpoint on the line you just added. Build and run the app, and wait for this breakpoint to hit. You should see the disassembly view again, but if you don't, use the **Always Show Disassembly** option.



As you've learned in the **Stack Related Opcodes** section, `call` is responsible for the execution of a function. Since there's only one `call` opcode between where `RIP` is right now and the end of `viewDidLoad`, this means this `call` must be the one responsible for calling `executeLotsOfArguments(one:two:three:four:five:six:seven:eight:nine:ten:)`.

But what are all the rest of the instructions before `call`? Let's find out.

These instructions set up the stack as necessary to pass the additional parameters. You have your usual 6 parameters being put into the appropriate registers, as seen by the instructions before where `RIP` is now, from `mov edx, 0x1` onwards.

But parameters 7 and beyond need to be passed on the stack. This is done with the following instructions:

```
0x1000013e2 <+178>: mov    qword ptr [rsp], 0x7
0x1000013ea <+186>: mov    qword ptr [rsp + 0x8], 0x8
0x1000013f3 <+195>: mov    qword ptr [rsp + 0x10], 0x9
0x1000013fc <+204>: mov    qword ptr [rsp + 0x18], 0xa
```

Looks scary, doesn't it? I'll explain.

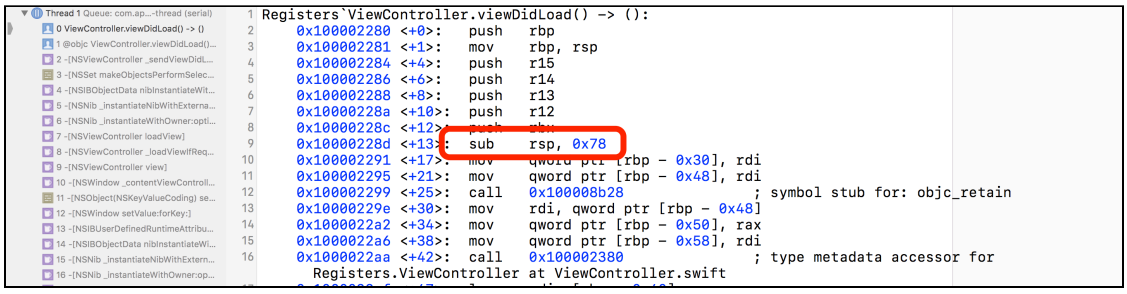
The brackets containing `RSP` and an optional value indicate a dereference, just like a `*` would in C programming. The first line above says "put `0x7` into the memory address pointed to by `RSP`." The second line says "put `0x8` into the memory address pointed to by `RSP` plus `0x8`." And so on.

This is placing values onto the stack. But take note the values are not explicitly pushed using the `push` instruction, which would decrease the `RSP` register. Why is that?

Well, as you've learned, during a `call` instruction the return address is pushed onto the stack. Then, in the function prologue, the base pointer is pushed onto the stack, and then the base pointer gets set to the stack pointer.

What you haven't learned yet is the compiler will actually make room on the stack for "scratch space". That is, the compiler allocates space on the stack for local variables in a function as necessary.

You can easily determine if extra scratch space is allocated for a stack frame by looking for the `sub rsp, VALUE` instruction in the function prologue. For example, click on the `viewDidLoad` stack frame and scroll to the top. Observe how much scratch space has been created:



The compiler has been a little bit clever here; instead of doing lots of pushes, it knows it has allocated some space on the stack for itself, and fills in values before the function call passing these extra parameters. Individual `push` instructions would involve more writes to `RSP`, which would be less efficient.

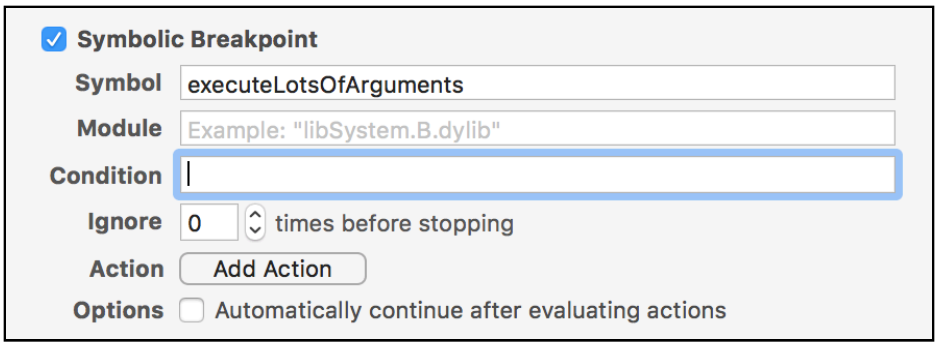
Time to look at this scratch space in more depth.

The stack and debugging info

The stack is not only used when calling functions, but it's also used as a scratch space for a function's local variables. Speaking of which, how does the debugger know which addresses to reference when printing out the names of variables that belong to that function?

Let's find out!

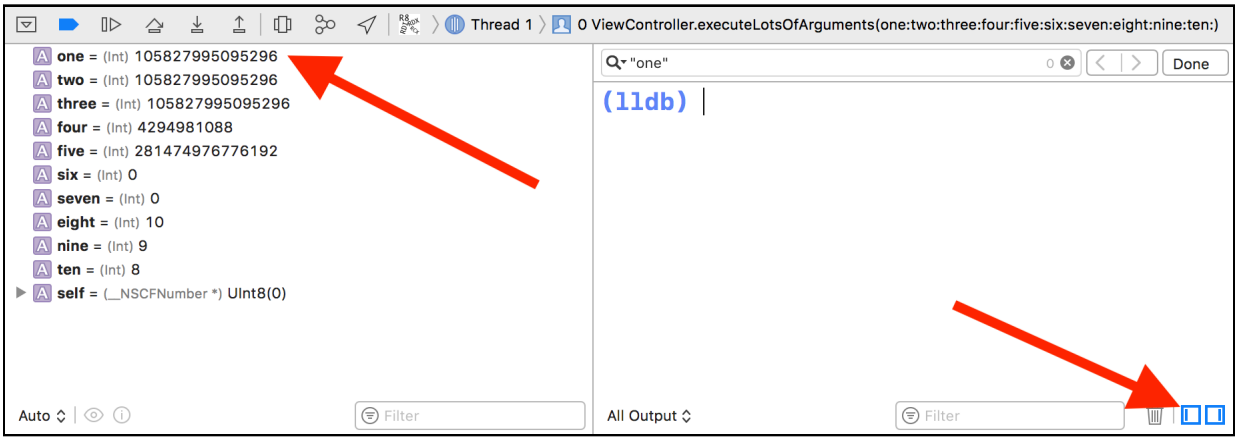
Clear all the breakpoints you've set and create a new Symbolic breakpoint on `executeLotsOfArguments`.



Build and run the app, then wait for the breakpoint to hit.

As expected, control should stop at the ever-so-short name of a function: `executeLotsOfArguments(one:two:three:four:five:six:seven:eight:nine:ten:)`, from here on, now referred to as `executeLotsOfArguments`, because its full name is a bit of a mouthful!

In the lower right corner of Xcode, click on **Show the Variables View**:



From there, look at the value pointed at by the `one` variable... it definitely ain't holding the value of `0x1` at the moment. This value seems to be gibberish!

Why is `one` referencing a seemingly random value?

The answer is stored by the **DWARF Debugging Information** embedded into the debug build of the Registers application. You can dump this information to help give you insight into what the `one` variable is referencing in memory.

In LLDB, type the following:

```
(lldb) image dump symfile Registers
```

You'll get a crazy amount of output. Search for (Cmd + F) the word "**one**"; include the quotes within your search.

Below is a (very) truncated output that includes the relevant information:

```
Swift.String, type_uid = 0x300000222
0x7f9b4633a988:     Block{0x300000222}, ranges =
[0x1000035e0-0x100003e7f)
0x7f9b48171a20:       Variable{0x30000023f}, name = "one", type =
{d50e000003000000} 0x00007f9b4828d2a0 (Swift.Int), scope = parameter,
decl = ViewController.swift:39, location =  DW_OP_fbreg(-32)
```

Based upon the output, the variable named `one` is of type **Swift.Int**, found in `executeLotsOfArguments`, whose location can be found at **DW_OP_fbreg(-32)**. This rather obfuscated code actually means base pointer minus 40, i.e. `RBP - 32`. Or in hexadecimal, `RBP - 0x20`.

This is important information. It tells the debugger the variable called `one` can always be found in this memory address. Well, not always, but always when that variable is valid, i.e. it's in scope.

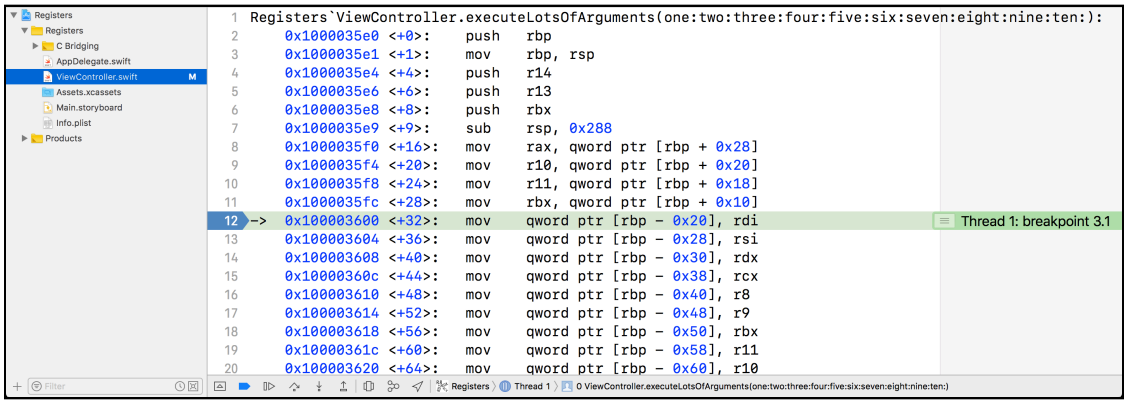
You may wonder why it can't just be `RDI`, since that's where the value is passed to the function, and it's also the first parameter. Well, `RDI` may need to be reused later on within the function, so using the stack is a safer bet for storage.

The debugger should still be stopped on `executeLotsOfArguments`. Make sure you're viewing the **Always Show Disassembly** output and hunt for the assembly:

```
mov    qword ptr [rbp - 0x20], rdi
```

Once you've found it in the assembly output of `executeLotsOfArguments`, create a breakpoint on this line of assembly.

Continue execution so LLDB will stop on this line of assembly.



Try printing out the output of `one` in LLDB:

```
(lldb) po one
```

Gibberish, still. Hmph.

Remember, `RDI` will contain the first parameter passed into the function. So to make the debugger be able to see the value that `one` should be, `RDI` needs to be written to the address where `one` is stored. In this case, `RBP - 0x20`.

Now, perform an assembly instruction step in LLDB:

```
(lldb) si
```

Print the value of `one` again.

```
(lldb) po one
```

Awwww.... yeah! It's working! The value `one` is referencing is correctly holding the value `0x1`.

You may be wondering what happens if `one` changes. Well, `RBP - 0x20` needs to change in that case too. This would potentially be another instruction needed to write it there as well as wherever the value is used. This is why debug builds are so much slower than release builds.

Stack exploration takeaways

Don't worry. This chapter is almost done. But there are some very important takeaways that should be remembered from your stack explorations.

Provided you're in a function, and the function has finished executing the function prologue, the following items will hold true to x64 assembly:

- `RBP` will point to the start of the stack frame for this function.
- `*RBP` will contain the address of the start of the previous stack frame. (Use `x/gx $rbp` in LLDB to see it).
- `*(RBP + 0x8)` will point to the return address to the previous function in the stack trace (Use `x/gx '$rbp + 0x8'` in LLDB to see it).
- `*(RBP + 0x10)` will point to the 7th parameter (if there is one).
- `*(RBP + 0x18)` will point to the 8th parameter (if there is one).
- `*(RBP + 0x20)` will point to the 9th parameter (if there is one).
- `*(RBP + 0x28)` will point to the 10th parameter (if there is one).
- `RBP – X` where `X` is multiples of `0x8`, will reference local variables to that function.

Where to go from here?

Now that you're familiar with the `RBP` and `RSP` registers, you've got a homework assignment!

Attach LLDB to a program (any program, source or no source) and traverse the stack frame using only the `RBP` register. Create a breakpoint on an easily triggerable method. One good example is `-[NSView hitTest:]`, if you attach to a macOS application such as Xcode, and click on a view.

It's important to ensure the breakpoint you choose to add is *not* a Swift function. You're going to inspect registers, — and recall you can't (easily) do this in the Swift context.

Once the breakpoint has been triggered, make sure you're on frame 0 by typing the following into LLDB:

```
(lldb) f 0
```

The `f` command is an alias for `frame select`.

You should see the following two instructions at the top of this function:

```
push   rbp
mov    rbp, rsp
```

These instructions form the start of the function prologue and push `RBP` onto the stack and then set `RBP` to `RSP`.

Step over both of these instructions using `si`.

Now the base pointer is set up for this stack frame, you can traverse the stack frames yourself by inspecting the base pointer.

Execute the following in LLDB:

```
(lldb) p uintptr_t $Previous_RBP = *(uintptr_t *)$rsp
```

So now `$Previous_RBP` equals the old `RBP`, i.e. the start of the stack frame from the function that called this one.

Recall the first thing on the stack frame is the address to where the function should return. So you can find out where the *previous* function will return to. This will therefore be where the debugger is stopped in Frame 2.

To find this out and check that you're right, execute the following in LLDB:

```
(lldb) x/gx '$Previous_RBP + 0x8'
```

This will print something like this:

```
0x7fff5fbfd718: 0x00007fffa83ed11b
```

Confirm this address equals the return address in Frame 1 with LLDB:

```
(lldb) f 2
```

It will look something like this, depending on what you decided to set the initial breakpoint in:

```
frame #2: 0x00007fffa83ed11b AppKit`-[NSWindow
_setFrameCommon:display:stashSize:] + 3234
AppKit`-[NSWindow _setFrameCommon:display:stashSize:]:
    0x7fffa83ed11b <+3234>: xor    ebx, ebx
    0x7fffa83ed11d <+3236>: mov    rsi, qword ptr [rip + 0x1c5a9d8c] ;
"_bindingAdaptor"
    0x7fffa83ed124 <+3243>: mov    rdi, r12
    0x7fffa83ed127 <+3246>: call   qword ptr [rip + 0x1c319f53] ; (void
*)0x00007fffbee77b40: objc_msgSend
```

The first address that it spits out should match the output of your earlier `x/gx` command.

Good luck and may the assembly be with you!

Section III: Low Level

With a foundation of assembler theory solidly below you, it's time to explore other aspects of how programs work. This section is an eclectic grab-bag of weird and fun studies into reverse engineering, seldom-used APIs and debugging strategies.

Chapter 14: Hello, Ptrace

Chapter 15: Dynamic Frameworks

Chapter 16: Hooking & Executing Code with dlopen & dlsym

Chapter 17: Exploring & Method Swizzling Objective-C Frameworks

Chapter 18: Hello, Mach-O

Chapter 19: Mach-O Fun

Chapter 20: Code Signing

Chapter 14: Hello, Ptrace

As alluded to in the introduction to this book, debugging is not entirely about just fixing stuff. Debugging is the process of gaining a better understanding of what's happening behind the scenes. In this chapter, you'll explore the foundation of debugging, namely, a system call responsible for a process attaching itself to another process: **`ptrace`**.

In addition, you'll learn some common security tricks developers use with `ptrace` to prevent a process from attaching to their programs. You'll also learn some easy workarounds for these developer-imposed restrictions.

System calls

Wait wait wait… `ptrace` is a system call. What's a *system* call?

A **system call** is a powerful, lower-level service provided by the kernel. System calls are the foundation for user-land frameworks, such as C's stdlib, Cocoa, UIKit, or even your own brilliant frameworks are built upon.

macOS Mojave Sierra has about 533 system calls. Open a Terminal window and run the following command to get a very close estimate of the number of systems calls available in your system:

```
sudo dtrace -ln 'syscall:::entry' | wc -l
```

This command uses an incredibly powerful tool named **DTrace** to inspect system calls present on your macOS machine.

> **Note**: Remember, you'll need to disable SIP (See Chapter 1) if you want to use DTrace. In addition, you'll also need `sudo` for the `DTrace` command since DTrace can monitor processes across multiple users, as well as perform some incredibly powerful actions. With great power comes great responsibility — that's why you need `sudo`.

You'll learn more about how to bend DTrace to your will in the 5th section of this book. For now you'll use simple DTrace commands to get system call information out of `ptrace`.

The foundation of attachment, ptrace

You're now going to take a look at the `ptrace` system call in more depth. Open a Terminal console. Before you start, make sure to clear the Terminal console by pressing `⌘ + K`. Next, execute the following DTrace inline script in Terminal to see how `ptrace` is called:

```
sudo dtrace -qn 'syscall::ptrace:entry { printf("%s(%d, %d, %d, %d) from
%s\n", probefunc, arg0, arg1, arg2, arg3, execname); }'
```

This creates a `DTrace` probe that will execute every time the `ptrace` function executes; it will spit out the arguments of the `ptrace` system call as well as the executable responsible for calling.

Don't worry about the semantics of this `DTrace` script; you'll become uncomfortably familiar with this tool in a later set of chapters. For now, just focus on what's returned from the Terminal.

Create a new tab in Terminal with the shortcut `⌘ + T`.

> **Note**: If you haven't disabled Rootless yet, you'll need to check out Chapter 1 for more information on how to disable it, or else `ptrace` will fail when attaching to Finder and your DTrace scripts will not work.

Once Rootless is disabled, type the following into the new Terminal tab:

```
lldb -n Finder
```

Once you've attached to the `Finder` application, the DTrace probe you set up on your first Terminal tab will spit out some information similar to the following:

```
ptrace(14, 459, 0, 0) from debugserver
```

It seems a process named **debugserver** is responsible for calling `ptrace` and attaching to the `Finder` process. But how was `debugserver` called? You attached to `Finder` using LLDB, not `debugserver`. And is this `debugserver` process even still alive?

Time to answer these questions. Create a new tab in Terminal (`⌘ + T`). Next, type the following into the Terminal window:

```
pgrep debugserver
```

Provided LLDB has attached successfully and is running, you'll receive an integer output representing `debugserver`'s process ID, or **PID**, indicating `debugserver` is alive and well and running on your computer.

Since `debugserver` is currently running, you can find out how `debugserver` was started. Type the following:

```
ps -fp `pgrep -x debugserver`
```

Be sure to note that the above commands uses *backticks*, not single quotes, to make the command work.

This will give you the full path to the location of `debugserver`, along with all arguments used to launch this process.

You'll see something similar to the following:

```
/Applications/Xcode-beta.app/Contents/SharedFrameworks/LLDB.framework/
Resources/debugserver --native-regs --setsid --reverse-connect
127.0.0.1:59297
```

Cool! This probably makes you wonder how the functionality changes when you subtract or modify certain launch arguments. For instance, what would happen if you got rid of `--reverse-connect 127.0.0.1:59297`?

So which process launched `debugserver`? Type the following:

```
ps -o ppid= $(pgrep -x debugserver)
```

This will dump out the parent PID responsible for launching `debugserver`. You'll get an integer similar to the following:

```
82122
```

As always when working with PIDs, they will very likely be different on your computer (and from run-to-run) than what you see here.

All right, numbers are interesting, but you're dying to know the actual name associated with this PID. You can get this information by executing the following in Terminal, replacing the number with the PID you discovered in the previous step:

```
ps -a 82122
```

You'll get the name, fullpath, and launch arguments of the process responsible for launching `debugserver`:

```
PID   TT  STAT      TIME COMMAND
82122 s000  S+     0:05.35 /Applications/Xcode.app/Contents/Developer/
usr/bin/lldb -n Finder
```

As you can see, `LLDB` was responsible for launching the debugserver process, which then attached itself to `Finder` using the `ptrace` system call. Now you know where this call is coming from, you can take a deeper dive into the function arguments passed into `ptrace`.

ptrace arguments

You're able to infer the process and arguments executed when `ptrace` was called. Unfortunately, they're just numbers, which are rather useless to you at the moment. It's time to make sense of these numbers using the **<sys/ptrace.h>** header file.

To do this, you'll use a macOS application to guide your understanding.

Open up the **helloptrace** application, which you'll find in the resources folder for this chapter. This is a macOS Terminal command application and is as barebones as they come. All it does is launch then complete with no output to `stdout` at all.

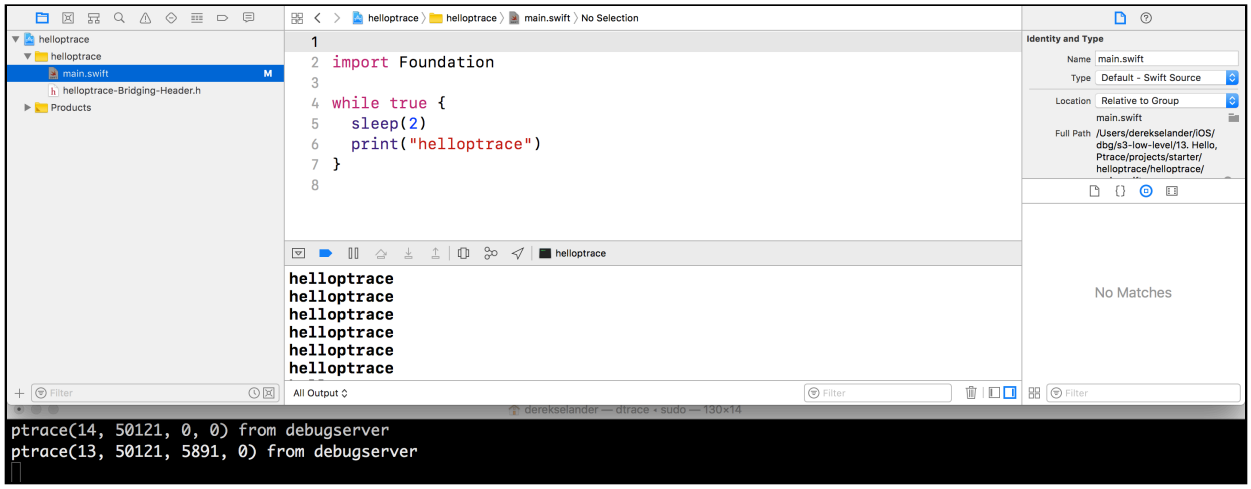
The only thing of interest in this project is a bridging header used to import the `ptrace` system call API into Swift.

Open **main.swift** and add the following code to the end of the file:

```
while true {
  sleep(2)
  print("helloptrace")
}
```

Next, position Xcode and the DTrace Terminal window so they are both visible on the same screen.

Build and run the application. Once your app has launched and `debugserver` has attached, observe the output generated by the DTrace script.



Take note of the DTrace Terminal window. Two new `ptrace` calls will happen when the `helloptrace` process starts running. The output of the DTrace script will look similar to this:

```
ptrace(14, 50121, 0, 0) from debugserver
ptrace(13, 50121, 5891, 0) from debugserver
```

Use Xcode's **Open Quickly** feature (`⌘ + Shift + O`) and type **/usr/include/sys/ptrace.h**.

A look in **ptrace.h** gives the following function prototype for `ptrace`:

```
int ptrace(int _request, pid_t _pid, caddr_t _addr, int _data);
```

The first parameter is what you want `ptrace` to do. The second parameter is the PID you want to act upon. The third and fourth parameters depend on the first parameter.

Take a look back at your earlier DTrace output. Your first line of output was something similar to the following:

```
ptrace(14, 50121, 0, 0) from debugserver
```

Compare the first parameter to **ptrace.h** header and you'll see the first parameter, `14`, actually stands for **`PT_ATTACHEXC`**. What does this `PT_ATTACHEXC` mean? To get information about this parameter, first, open a Terminal window. Finally, type **man ptrace** and search for `PT_ATTACHEXC`.

> **Note**: You can perform case-sensitive searches on `man` pages by pressing `/`, followed by your search query. You can search downwards to the next hit by pressing **N** or upwards to the previous hit by pressing **Shift** + **N**.

You'll find some relevant info about `PT_ATTACHEXC` with the following output obtained from the `ptrace` `man` page:

```
This request allows a process to gain control of an otherwise unrelated
process and begin tracing it. It does not need any cooperation from the
to-be-traced process. In this case, `pid` specifies the process ID of the
to-be-traced process, and the other two arguments are ignored.
```

With this information, the reason for the first call of `ptrace` should be clear. This call says "hey, attach to this process", and attaches to the process provided in the second parameter.

Onto the next `ptrace` call from your DTrace output:

```
ptrace(13, 50121, 5891, 0) from debugserver
```

This one is a bit trickier to understand, since Apple decided to not give any `man` documentation about this one. This call relates to the internals of a process attaching to another one.

If you look at the `ptrace` API header, 13 stands for **`PT_THUPDATE`** and relates to how the controlling process, in this case, `debugserver`, handles UNIX signals and Mach messages passed to the controlled process; in this case, `helloptrace`.

The kernel needs to know how to handle signal passing from a process controlled by another process, as in the `Signals` project from Section 1. The controlling process could say it doesn't want to send any signals to the controlled process.

This specific `ptrace` action is an implementation detail of how the Mach kernel handles `ptrace` internally; there's no need to dwell on it.

Fortunately, there are other *documented* signals definitely worth exploring through `man`. One of them is the **`PT_DENY_ATTACH`** action, which you'll learn about now.

Creating attachment issues

A process can actually specify it doesn't want to be attached to by calling `ptrace` and supplying the `PT_DENY_ATTACH` argument. This is often used as an anti-debugging mechanism to prevent unwelcome reverse engineers from discovering a program's internals.

You'll now experiment with this argument. Open **main.swift** and add the following line of code before the `while` loop:

```
ptrace(PT_DENY_ATTACH, 0, nil, 0)
```

Build and run, keep on eye on the debugger console and see what happens.

The program will exit and output the following to the debugger console:

```
Program ended with exit code: 45
```

> **Note**: You may need to open up the debug console by clicking **View ▸ Debug Area ▸ Activate Console** (or `⌘ + Shift + Y` if you're one of those cool, shortcut devs) to see this.

This happened because Xcode launches the `helloptrace` program by default with LLDB automatically attached. If you execute the `ptrace` function with `PT_DENY_ATTACH`, LLDB will exit early and the program will stop executing.

If you were to try and execute the `helloptrace` program, and tried later to attach to it, LLDB would fail in attaching and the `helloptrace` program would happily continue execution, oblivious to `debugserver`'s attachment issues.

There are numerous macOS (and iOS) programs that perform this very action in their production builds. However, it's rather trivial to circumvent this security precaution. Ninja debug mode activated!

Getting around PT_DENY_ATTACH

Once a process executes `ptrace` with the `PT_DENY_ATTACH` argument, making an attachment greatly escalates in complexity. However, there's a *much* easier way of getting around this problem.

Typically a developer will execute `ptrace(PT_DENY_ATTACH, 0, 0, 0)` somewhere in the main executable's code — oftentimes, right in the main function.

Since LLDB has the `–w` argument to wait for the launching of a process, you can use LLDB to "catch" the launch of a process and perform logic to augment or ignore the `PT_DENY_ATTACH` command before the process has a chance to execute `ptrace`!

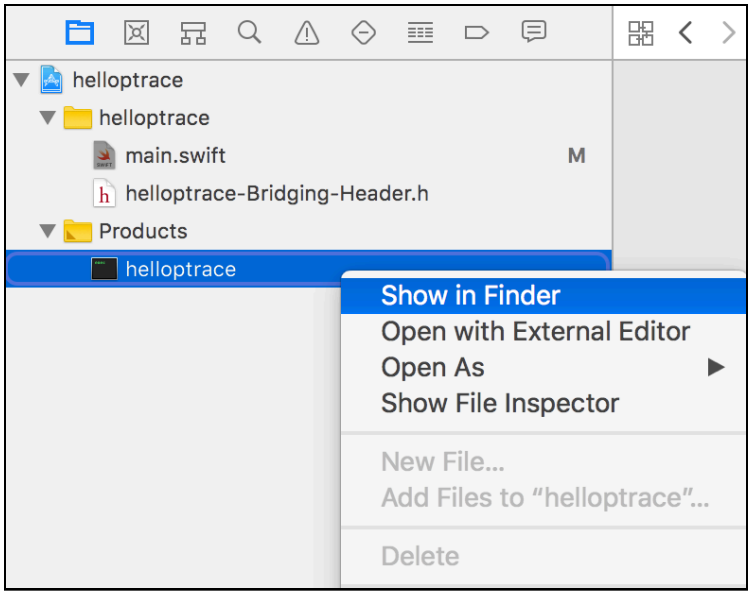
Open a new Terminal window and type the following:

```
sudo lldb -n "helloptrace" -w
```

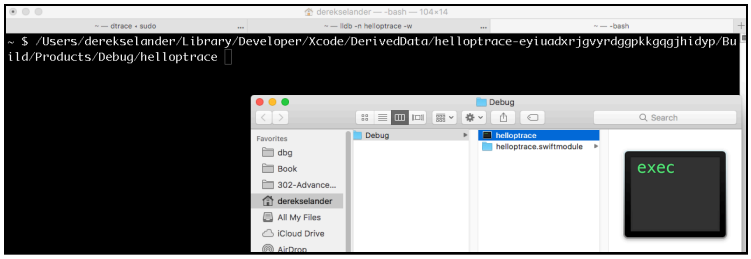
This starts an `lldb` session and attaches to the `helloptrace` program, but this time `–w` tells `lldb` to wait until a new process with the name `helloptrace` has started.

You need to use `sudo` due to an ongoing bug with LLDB and macOS security when you tell LLDB to wait for a Terminal program to launch.

In the Project Navigator, open the **Products** folder and right click on the **helloptrace** executable. Next, select **Show in Finder**.



Next, drag the `helloptrace` executable into a new Terminal tab. Finally, press **Enter** to start the executable.



Now, open the previously created Terminal tab, where you had LLDB sit and wait for the `helloptrace` executable.

If everything went as expected, LLDB will see `helloptrace` has started and will launch itself, attaching to this newly created `helloptrace` process.

In LLDB, create the following regex breakpoint to stop on any type of function containing the word `ptrace`:

```
(lldb) rb ptrace -s libsystem_kernel.dylib
```

This will add a breakpoint on the userland gateway to the actual kernel `ptrace` function. Next, type `continue` into the Terminal window.

```
(lldb) continue
```

You'll break right before the `ptrace` function is about to be executed. However, you can simply use LLDB to return early and not execute that function. Do that now like so:

```
(lldb) thread return 0
```

Next, simply just continue:

```
(lldb) continue
```

Although the program entered the `ptrace` userland gateway function, you told LLDB to return early and not execute the logic that will execute the kernel `ptrace` system call.

Navigate to the `helloptrace` output tab and verify it's outputting "`helloptrace`" over and over. If so, you've successfully bypassed `PT_DENY_ATTACH` and are running LLDB while still attached to the `helloptrace` command!

In a couple chapters, you'll explore an alternative method to crippling external functions like `ptrace` by inspecting Mach-O's `__DATA.__la_symbol_ptr` section along with the lovely `DYLD_INSERT_LIBRARIES` environment variable.

Other anti-debugging techniques

Since we're on the topic of anti-debugging, let's put iTunes on the spot: for the longest time, iTunes actually used the `ptrace`'s `PT_DENY_ATTACH`. However, the current version of iTunes (12.7.0 at the time of writing) has opted for a different technique to prevent debugging.

iTunes will now check if it's being debugged using the powerful **sysctl** function, then kill itself if true. `sysctl` is another kernel function (like `ptrace`) that gets or sets kernel values. iTunes repeatedly calls `sysctl` while it's running using a `NSTimer` to call out to the logic.

Below is a simplified code example in Swift of what iTunes is doing:

```
let mib = UnsafeMutablePointer<Int32>.allocate(capacity: 4)
mib[0] = CTL_KERN
mib[1] = KERN_PROC
mib[2] = KERN_PROC_PID
mib[3] = getpid()

var size: Int = MemoryLayout<kinfo_proc>.size
var info: kinfo_proc? = nil

sysctl(mib, 4, &info, &size, nil, 0)

if (info.unsafelyUnwrapped.kp_proc.p_flag & P_TRACED) > 0 {
  exit(1)
}
```

I am not going to go into the details of the expected params for `sysctl` yet, we'll save that for a different chapter. Just know that there is more than one way to skin a cat.

Where to go from here?

With the DTrace dumping script you used in this chapter, explore parts of your system and see when `ptrace` is called.

If you're feeling cocky, read up on the `ptrace` `man` pages and see if you can create a program that will automatically attach itself to another program on your system.

Still have energy? Go `man sysctl`. That will be some good night-time reading.

Remember, having attachment issues is not always a bad thing!

Chapter 15: Dynamic Frameworks

If you've developed any type of Apple GUI software, you've definitely used dynamic frameworks in your day-to-day development.

A dynamic framework is a bundle of code loaded into an executable at runtime, instead of at compile time. Examples in iOS include `UIKit` and the `Foundation` frameworks. Frameworks such as these contain a dynamic library and optionally assets, such as images.

There are numerous advantages in electing to use dynamic frameworks instead of static frameworks. The most obvious advantage is you can make updates to the framework without having to recompile the executable that depends on the framework.

Imagine if, for every major or minor release of iOS, Apple said, "Hey y'all, we need to update `UIKit` so if you could go ahead and update your app as well, that would be grrrreat." There would be blood in the streets and the only competition would be Android vs. Windows Phone!

Why dynamic frameworks?

In addition to the positives of using dynamic frameworks, the kernel can map the dynamic framework to multiple processes that depend on the framework. Take `CFNetwork`, for example: it would be stupid and a waste of disk space if each running iOS app kept a unique copy of `CFNetwork` resident in memory. Furthermore, there could be different versions of `CFNetwork` compiled into each app, making it incredibly difficult to track down bugs.

As of iOS 8, Apple decided to lift the dynamic library restriction and allow third-party dynamic libraries to be included in your app. The most obvious advantage was that developers could share frameworks across different iOS extensions, such as the Today Extension and Action Extensions.

Today, all Apple platforms allow third party dynamic frameworks to be included without rejection in the ever-so-lovely Apple Review process.

With dynamic frameworks comes a very interesting aspect of learning, debugging, and reverse engineering. Since you've the ability to load the framework at runtime, you can use LLDB to explore and execute code at runtime, which is great for spelunking in both public and private frameworks.

Statically inspecting an executable's frameworks

Compiled into each executable is a list of dynamic libraries (most often, frameworks), expected to be loaded at runtime. This can be further broken down into a list of required frameworks and a list of optional frameworks. The loading of these dynamic libraries into memory is done using a special framework called the **dynamic loader**, or `dyld`.

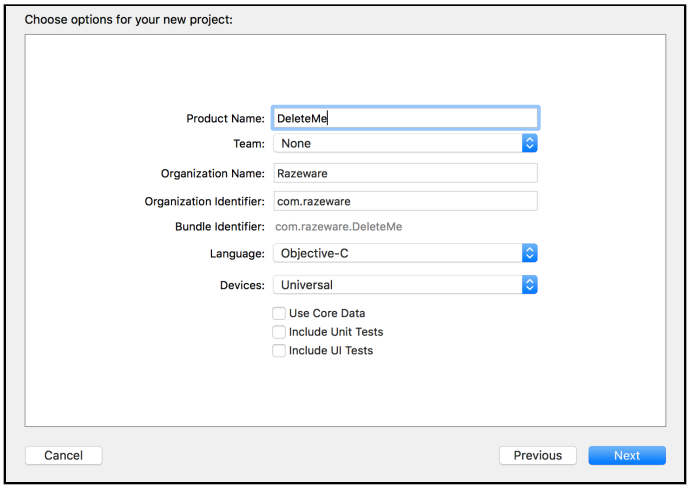
If a required framework fails to load, the dynamic library loader will kill the program. If an optional framework fails to load, everything continues as usual, but code from that library will obviously not be able to run!

You may have used the optional framework feature in the past, perhaps when your iOS or Mac app needed to use code from a library added in a newer OS version than the version targeted by your app. In such cases, you'd perform a runtime check around calls to code in the optional library to check if the library was loaded.

I spout tons of this theory stuff, but it'll make more sense if you see it for yourself.

Open Xcode and create a new iOS project, **Single View Application** named **DeleteMe**. Yep, this project won't hang around for long, so feel free to remove it once you're done with this chapter.

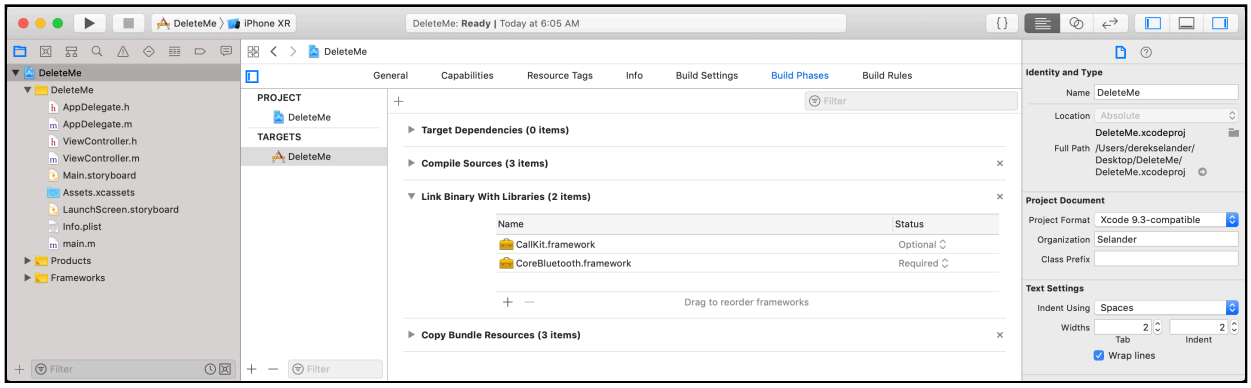
You'll not write a line of code within the app (but within the load commands is a different story). Make sure you choose Objective-C then click `Next`.



> **Note**: You're using Objective-C because there's more going on under the hood in a Swift app. At the time of writing, the Swift ABI is not finalized, so every method Swift uses to bridge Objective-C uses a dynamic framework packaged into your app to "jump the gap" to Objective-C. This means within the Swift bridging frameworks are the corresponding dependencies to the proper Objective-C Frameworks. For example, **libswiftUIKit.dylib** will have a required dependency on the `UIKit` framework.

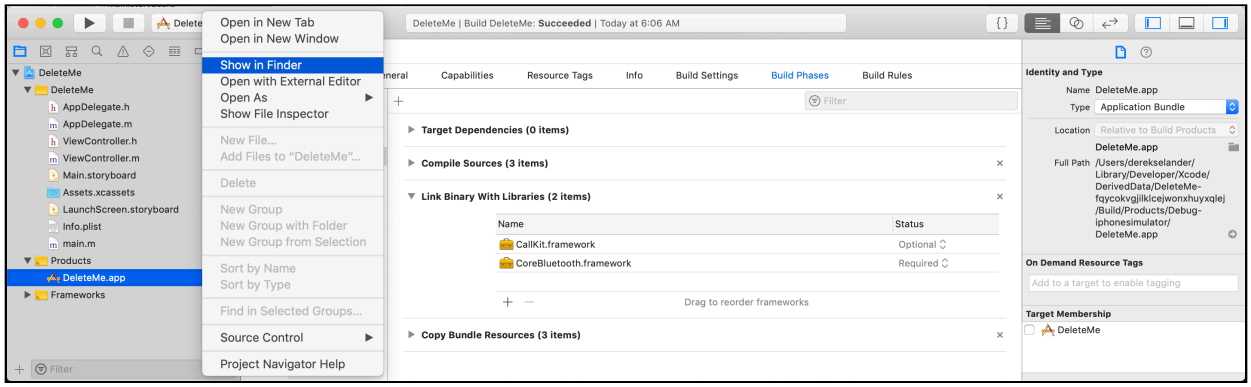
Click on the Xcode project at the top of the project navigator. Then click on the **DeleteMe** target. Next, click on the **Build Phases** and open up the **Link Binary With Libraries**.

Add the **CoreBluetooth** and **CallKit** framework. To the right of the CallKit framework, select **Optional** from the drop-down. Ensure that the CoreBluetooth framework has the **Required** value set as shown below.

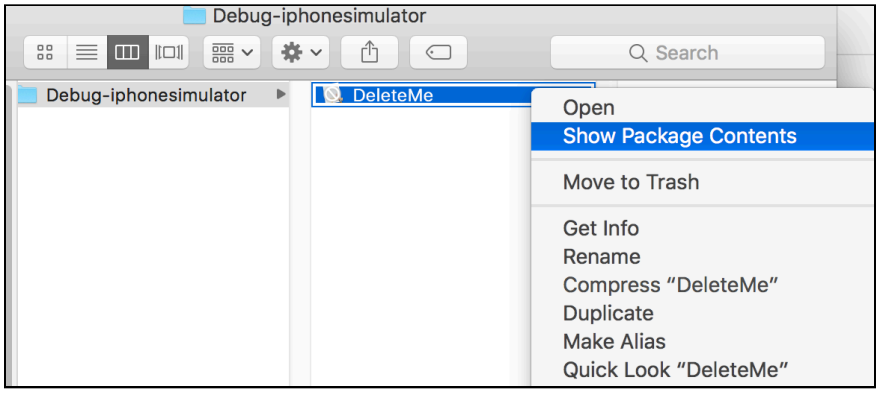


Build the project on the simulator using **Cmd + B**. **Do not run just yet**. Once the project has been successfully built for the simulator, open the **products** directory in the Xcode project navigator.

Right click on the produced executable, `DeleteMe`, and select **Show in Finder**.



Next, open up the `DeleteMe` IPA by right clicking the IPA and selecting **Show Package Contents**.



Next, open a new `Terminal` window and type the following but **don't press Enter**:

```
otool -L
```

Be sure to add a space at the end of the command. Next, drag the `DeleteMe` executable from the `Finder` window into the `Terminal` window. When finished, you should have a command that looks similar to the following:

```
otool -L /Users/derekselander/Library/Developer/Xcode/DerivedData/
DeleteMe-fqycokvgjilklcejwonxhuyxqlej/Build/Products/Debug-
iphonesimulator/DeleteMe.app/DeleteMe
```

Press Enter and observe the output. You'll see something similar to the following:

```
/System/Library/Frameworks/CallKit.framework/CallKit (compatibility
version 1.0.0, current version 1.0.0)

/System/Library/Frameworks/CoreBluetooth.framework/CoreBluetooth
(compatibility version 1.0.0, current version 1.0.0)

/System/Library/Frameworks/Foundation.framework/Foundation (compatibility
version 300.0.0, current version 1556.0.0)

/usr/lib/libobjc.A.dylib (compatibility version 1.0.0, current version
228.0.0)

/usr/lib/libSystem.B.dylib (compatibility version 1.0.0, current version
1252.200.5)

/System/Library/Frameworks/UIKit.framework/UIKit (compatibility version
1.0.0, current version 61000.0.0)
```

You found the compiled binary `DeleteMe` and dumped out the list of dynamic frameworks it links to using the ever-so-awesome **`otool`**. Take note of the instructions to `CallKit` and the `CoreBluetooth` framework you manually added earlier. By default, the compiler automatically adds the "essential" frameworks to the iOS app, like `UIKit` and `Foundation`.

Take note of the directory path responsible for loading these frameworks:

```
/System/Library/Frameworks/
/usr/lib/
```

Remember these directories; you'll revisit them for a "eureka" moment later on.

Let's go a tad bit deeper. Remember how you optionally required the `CallKit` framework, and required the `CoreBluetooth` framework? You can view the results of these decisions by using `otool`.

In `Terminal`, press the up arrow to recall the previous `Terminal` command. Next, change the capital **L** to a lowercase **l** and press Enter. You'll get a longer list of output that shows all the **load commands** for the `DeleteMe` executable.

```
otool -l /Users/derekselander/Library/Developer/Xcode/DerivedData/
DeleteMe-fqycokvgjilklcejwonxhuyxqlej/Build/Products/Debug-
iphonesimulator/DeleteMe.app/DeleteMe
```

Search for load commands pertaining to `CallKit` by pressing **Cmd + F** and typing **CallKit**. You'll stumble across a load command similar to the following:

```
Load command 12
          cmd LC_LOAD_WEAK_DYLIB
      cmdsize 80
         name /System/Library/Frameworks/CallKit.framework/CallKit
(offset 24)
   time stamp 2 Wed Dec 31 17:00:02 1969
      current version 1.0.0
compatibility version 1.0.0
```

Next, search for the `CoreBluetooth` framework as well:

```
Load command 13
          cmd LC_LOAD_DYLIB
      cmdsize 96
         name /System/Library/Frameworks/CoreBluetooth.framework/
CoreBluetooth (offset 24)
   time stamp 2 Wed Dec 31 17:00:02 1969
      current version 1.0.0
compatibility version 1.0.0
```

Compare the **cmd** in the load commands output. In `CallKit`, the load command is `LC_LOAD_WEAK_DYLIB`, which represents an optional framework, while the `LC_LOAD_DYLIB` of the `CoreBluetooth` load command indicates a required framework.

This is ideal for an application that supports multiple iOS versions. For example, if you supported iOS 9 and up, you would strongly link the `CoreBluetooth` framework and weak link the `CallKit` framework since it's only available in iOS 10 and up.

Modifying the load commands

There's a nice little command that lets you augment and add the framework load commands named **`install_name_tool`**.

Open Xcode and build and run the application so the simulator is running `DeleteMe`. Once running, pause execution and in the LLDB `Terminal`, verify the `CallKit` framework is loaded into the `DeleteMe` address space. Pause the debugger, then type the following into LLDB:

```
(lldb) image list CallKit
```

If the `CallKit` module is correctly loaded into the process space, you'll get output similar to the following:

```
[  0] 0484D8BA-5CB8-3DD3-8136-D8A96FB7E15B 0x0000000102d10000 /
Applications/Xcode.app/Contents/Developer/Platforms/
iPhoneSimulator.platform/Developer/SDKs/iPhoneSimulator.sdk/System/
Library/Frameworks/CallKit.framework/CallKit
```

Time to hunt down where the `DeleteMe` application is running from. Open a new Terminal window and type the following:

```
pgrep -fl DeleteMe
```

Provided `DeleteMe` is running, this will give you the full path of `DeleteMe` under the simulator app. You'll get output similar to the following:

```
61175 /Users/derekselander/Library/Developer/CoreSimulator/Devices/
D0576CB9-42E1-494B-B626-B4DB75411700/data/Containers/Bundle/Application/
474C8786-CC4F-4615-8BB0-8447DC9F82CA/DeleteMe.app/DeleteMe
```

You'll now modify this executable's load commands to point to a different framework.

Grab the fullpath to the `DeleteMe` executable and assign it to a Terminal variable called **`app`**, like so:

```
app=/Users/derekselander/Library/Developer/CoreSimulator/Devices/
D0576CB9-42E1-494B-B626-B4DB75411700/data/Containers/Bundle/Application/
474C8786-CC4F-4615-8BB0-8447DC9F82CA/DeleteMe.app/DeleteMe
```

While you're at it, assign the **`CK`** and **`NC`** Terminal variables to the respective framework paths as well, like so:

```
CK=/System/Library/Frameworks/CallKit.framework/CallKit
NC=/System/Library/Frameworks/NotificationCenter.framework/
NotificationCenter
```

Stop the execution of the `DeleteMe` executable and temporarily close Xcode. If you were to accidentally build and run the `DeleteMe` application through Xcode at a later time, it would undo any tweaks you're about to make.

In the same `Terminal` window, use the **install_name_tool** command, along with the three newly-created Terminal variables, to change around the CallKit load command to call the `NotificationCenter` framework.

```
install_name_tool -change "$CK" "$NC" "$app"
```

If this were an app on a real iOS device, this would actually fail to run since this is invalidating the app's code signature. You have made changes to the application without resigning it, which breaks the cryptographic seal. Fortunately, this is an iOS Simulator app, so the rules are not as strict. You'll explore code signing further in the last chapter of this section.

Verify if your changes were actually applied:

```
otool -L "$app"
```

If everything went smoothly, you'll notice something different about the linked frameworks now:

```
/System/Library/Frameworks/NotificationCenter.framework/
NotificationCenter (compatibility version 1.0.0, current version 1.0.0)

/System/Library/Frameworks/CoreBluetooth.framework/CoreBluetooth
(compatibility version 1.0.0, current version 1.0.0)

/System/Library/Frameworks/Foundation.framework/Foundation (compatibility
version 300.0.0, current version 1556.0.0)

/usr/lib/libobjc.A.dylib (compatibility version 1.0.0, current version
228.0.0)

/usr/lib/libSystem.B.dylib (compatibility version 1.0.0, current version
1252.200.5)

/System/Library/Frameworks/UIKit.framework/UIKit (compatibility version
1.0.0, current version 61000.0.0)
```

Verify these changes exist at runtime.

You're in a bit of a predicament here. If you were to build and run a new version of `DeleteMe` using Xcode, it would erase these changes. Instead, launch the `DeleteMe` application through the simulator and then attach to it in a new LLDB `Terminal` window. To do this, launch `DeleteMe` in the simulator. Next, type the following into Terminal:

```
lldb -n DeleteMe
```

In LLDB, check if the `CallKit` framework is still loaded.

```
(lldb) image list CallKit
```

You'll get an error as output:

```
error: no modules found that match 'CallKit'
```

Can you guess what you'll do next? Yep! Verify the `NotificationCenter` framework is now loaded.

```
(lldb) image list NotificationCenter
```

Boom! You'll get output similar to the following:

```
[  0] 0FCE1DF5-7BAC-3195-94CB-C6100116FF99 0x000000010b8c7000 /
Applications/Xcode.app/Contents/Developer/Platforms/
iPhoneSimulator.platform/Developer/SDKs/iPhoneSimulator.sdk/System/
Library/Frameworks/NotificationCenter.framework/NotificationCenter
```

Changing around frameworks (or adding them!) to an already compiled binary is cool, but that took a little bit of work to set up. Fortunately, LLDB is wonderful for loading frameworks into a process at runtime, which is what you'll do next. Keep that LLDB Terminal session alive, because you'll learn about a much easier way to load in frameworks.

Loading frameworks at runtime

Before you get into the fun of learning how to load and explore commands at runtime, let me give you a command to help explore directories using LLDB. Start by adding the following to your `~/.lldbinit` file:

```
command regex ls 's/(.+)/po @import Foundation; [[NSFileManager
defaultManager] contentsOfDirectoryAtPath:@"%1" error:nil]/'
```

This creates a command named `ls`, which will take the directory path you give it and dump out the contents. This command will work on the directory of the device that's being debugged. For example, since you're running on the simulator on your computer's local drive it will dump that directory. If you were to run this on an attached iOS, tvOS or other appleOS device, it would dump the directory you give it on that device, with one minor caveat which you'll learn about shortly.

Since LLDB is already running and attached to `DeleteMe`, you'll need to load this command into LLDB manually as well since LLDB has already read the `~/.lldbinit` file. Type the following into your LLDB session:

```
(lldb) command source ~/.lldbinit
```

This simply reloads your `lldbinit` file.

Next, find the full path to the frameworks directory in the simulator by typing the following:

```
(lldb) image list -d UIKit
```

This will dump out the directory holding `UIKit`.

```
[  0] /Applications/Xcode.app/Contents/Developer/Platforms/
iPhoneSimulator.platform/Developer/SDKs/iPhoneSimulator.sdk//System/
Library/Frameworks/UIKit.framework
```

You actually want to go one level higher to the `Frameworks` directory. Copy that full directory path and use the new command `ls` that you just created, like so:

```
(lldb) ls /Applications/Xcode.app/Contents/Developer/Platforms/
iPhoneSimulator.platform/Developer/SDKs/iPhoneSimulator.sdk//System/
Library/Frameworks/
```

This will dump all the *public* frameworks available to the simulator. There are many more frameworks to be found in different directories, but you'll start here first.

From the list of frameworks, load the **Speech** framework into the `DeleteMe` process space like so:

```
(lldb) process load /Applications/Xcode.app/Contents/Developer/Platforms/
iPhoneSimulator.platform/Developer/SDKs/iPhoneSimulator.sdk//System/
Library/Frameworks/Speech.framework/Speech
```

LLDB will give you some happy output saying the `Speech` framework was loaded into your process space. Yay!

Here's something even cooler. By default, `dyld` will search a set of directories if it can't find the location of the framework. You don't need to specify the full path to the framework, just the framework library along with the framework's name.

Try this out by loading the `MessagesUI` framework.

```
(lldb) process load MessageUI.framework/MessageUI
```

You'll get the following output:

```
Loading "MessageUI.framework/MessageUI"...ok
Image 1 loaded.
```

Sweet.

Exploring frameworks

One of the foundations of reverse engineering is exploring dynamic frameworks. Since a dynamic framework requires the code to compile the binary into a **position independent** executable, you can still query a significant amount of information in the dynamic framework — even when the compiler strips the framework of debugging symbols. The binary needs to use position-independent code because the compiler doesn't know exactly where the code will reside in memory once `dyld` has done its business.

Having solid knowledge of how an application interacts with a framework can also give you insight into how the application works itself. For example, if a stripped application is using a `UITableView`, I'll set breakpoint queries in certain methods in `UIKit` to determine what code is responsible for the `UITableViewDataSource`.

Often when I'm exploring a dynamic framework, I'll simply load it into the processes address space and start running various `image lookup` queries (or my custom LLDB `lookup` command available at https://github.com/DerekSelander/lldb) to see what the module holds.

From there, I'll execute various interesting methods that look like they'd be fun to play around with.

Here's a nice little LLDB command regex you might want to stick into your `~/.lldbinit` file. It dumps Objective-C easily accessible class methods (i.e. Singletons) for exploration.

Add the following to your `~/.lldbinit` file.

```
command regex dump_stuff "s/(.+)/image lookup -rn '\+\[\w+(\(\w+\))?\ \w+
\]$' %1 /"
```

This command, `dump_stuff`, expects a framework or frameworks as input and will dump Objective-C class methods that have zero arguments. This definitely isn't a catch-all for all Objective-C naming conventions, but is a nice, simple command to use for a quick first pass when exploring a framework.

Load this command into the active LLDB session and then give it a go with the framework.

```
(lldb) command source ~/.lldbinit
(lldb) dump_stuff CoreBluetooth
```

You might find some amusing methods to play around with in the output…

If you jumped chapters and have that clueless face going on for the `image lookup` command, check out Chapter 7, "Image". You will add some helper LLDB `command regex`'s from the private introspection methods found in that chapter.

Add the following commands to your `~/.lldbinit` file as well:

```
command regex ivars 's/(.+)/expression -lobjc -0 -- [%1
_ivarDescription]/'
```

This will dump all the ivars of a inherited `NSObject` instance.

```
command regex methods 's/(.+)/expression -lobjc -0 -- [%1
_shortMethodDescription]/'
```

This will dump all the methods implemented by the inherited `NSObject` instance, or the class of the `NSObject`.

```
command regex lmethods 's/(.+)/expression -lobjc -0 -- [%1
_methodDescription]/'
```

This will recursively dump all the methods implemented by the inherited NSObject and recursively continue on to its superclass.

Using these commands it's quite easy to load, scan, and inspect interesting classes from different frameworks.

For example, you might choose to inspect classes found in the **UIPhotos** Framework. You can do the following:

```
(lldb) process load PhotosUI.framework/PhotosUI
```

From there, dump the class methods with no arguments:

```
(lldb) dump_stuff PhotosUI
```

Explore the methods and ivars found in the `PUScrubberSettings` class:

```
(lldb) ivars [PUScrubberSettings sharedInstance]
<PUScrubberSettings: 0x7ffb12818fb0>:
in PUScrubberSettings:
  _usePreviewScrubberMargins (BOOL): NO
  _useTrianglePositionIndicator (BOOL): NO
  _useSmoothingAnimation (BOOL): NO
  _dynamicSeekTolerance (BOOL): YES
  _previewInteractiveLoupeBehavior (unsigned long): 2
  _interactiveLoupeBehavior (unsigned long): 0
  _tapAnimationDuration (double): 0.5
...
```

Or perhaps you're just curious about what dynamic methods this class implements:

```
(lldb) methods PUScrubberSettings
<PUScrubberSettings: 0x11f80fc48>:
in PUScrubberSettings:
  Class Methods:
    + (id) sharedInstance; (0x11f57092b)
    + (id) settingsControllerModule; (0x11f570be8)
  Properties:
    @property (nonatomic) unsigned long previewInteractiveLoupeBehavior;
(@synthesize previewInteractiveLoupeBehavior =
_previewInteractiveLoupeBehavior;)
    @property (nonatomic) BOOL usePreviewScrubberMargins;  (@synthesize
usePreviewScrubberMargins = _usePreviewScrubberMargins;)
```

Or get *all* the methods available through this class and superclasses:

```
(lldb) lmethods PUScrubberSettings
```

Note: You only explored the frameworks in the public frameworks directory **System/Library/Frameworks**. There are many other fun frameworks to explore in other subdirectories starting in **System/Library**. For example, you'll find some entertainment in **System/Library/PrivateFrameworks**

Loading frameworks on an actual iOS device

If you have a valid iOS developer account, an application you've written, and a device, you can do the same thing you did on the simulator but on the device. The only difference is the location of the **System/Library** path.

If you're running an app on the simulator, the public frameworks directory will be located at the following location:

```
/Applications/Xcode.app/Contents/Developer/Platforms/
iPhoneSimulator.platform/Developer/SDKs/iPhoneSimulator.sdk/System/
Library/Frameworks/
```

But some super-observant readers might say, "Wait a second, using `otool -L` on the simulator gave us **/System/Library/Frameworks** as the absolute path, not that big long path above. What gives?"

Remember how I said `dyld` searches a specific set of directories for these frameworks? Well, there's a special simulator-specific version named `dyld_sim`, which looks up the proper simulator location. This is the correct path where these frameworks reside on an actual iOS device.

So if you're running on an actual iOS device, the frameworks path will be located at:

```
/System/Library/Frameworks/
```

But wait, I hear some others say, "What about sandboxing?"

The iOS kernel has different restrictions for different directory locations. In iOS 12 and earlier, the **/System/Library/** directory is readable by your process!

This makes sense because your process needs to call the appropriate public and private frameworks from within the processes address space.

If the Sandbox restricted reading of these directories, then the app wouldn't be able to load them in and then the app would fail to launch.

You can try this out by getting Xcode up and running and attached to any one of your iOS applications. While LLDB is attached to an iOS device, try running `ls` on the root directory:

```
(lldb) ls /
```

Now try the /System/Library/ directory:

```
(lldb) ls /System/Library/
```

Some directories will fail to load. This is the kernel saying "Nope!" However, some directories can be dumped.

You have the power to look at live frameworks and dynamically load them inside your app so you can play with and explore them. There are some interesting and powerful frameworks hidden in the **/System/Library** subdirectories for you to explore on your iOS, tvOS or watchOS device.

Where to go from here?

That **/System/Library** directory is really something. You can spend a lot of time exploring the different contents in that subdirectory. If you have an iOS device, go explore it!

In this chapter you learned how to load and execute frameworks through LLDB. However, you've been left somewhat high and dry for figuring out how to develop with dynamically loaded private frameworks in code.

In the next two chapters, you'll explore loading frameworks at runtime through code using Objective-C's method swizzling, as well as function interposition, which is a more Swifty-style strategy for changing around methods at runtime.

This is especially useful if you were to pull in a private framework. I think it's one of the most exciting things about reverse engineering Apple software.

Chapter 16: Hooking & Executing Code with dlopen & dlsym

Using LLDB, you've seen how easy it is to create breakpoints and inspect things of interest. You've also seen how to create classes you wouldn't normally have access to. Unfortunately, you've been unable to wield this power at development time because you can't get a public API if the framework, or any of its classes or methods, are marked as private. However, all that is about to change.

It's time to learn about the complementary skills of developing with these frameworks. In this chapter, you're going to learn about methods and strategies to "hook" into Swift and C code as well as execute methods you wouldn't normally have access to while developing.

This is a critical skill to have when you're working with something such as a private framework and want to execute or augment existing code within your own application. To do this, you're going to call on the help of two awesome functions: **`dlopen`** and **`dlsym`**.

The Objective-C runtime vs. Swift & C

Objective-C, thanks to its powerful runtime, is a truly dynamic language. Even when compiled and running, not even the program knows what will happen when the next `objc_msgSend` comes up.

There are different strategies for hooking into and executing Objective-C code; you'll explore these in the next chapter. This chapter focuses on how to hook into and use these frameworks under Swift.

Swift acts a lot like C or C++. If it doesn't need the dynamic dispatch of Objective-C, the compiler doesn't have to use it. This means when you're looking at the assembly for a Swift method that doesn't need dynamic dispatch, the assembly can simply call the address containing the method. This "direct" function calling is where the `dlopen` and `dlsym` combo really shines. This is what you're going to learn about in this chapter.

Setting up your project

For this chapter, you're going to use a starter project named **Watermark**, located in the **starter** folder.

This project is very simple. All it does is display a watermarked image in a `UIImageView`.



However, there's something special about this watermarked image. The actual image displayed is hidden away in an array of bytes compiled into the program. That is, the image is not bundled as a separate file inside the application. Rather, the image is actually located within the executable itself. Clearly the author didn't want to hand out the original image, anticipating people would reverse engineer the **Assets.car** file, which typically is a common place to hold images within an application. Instead, the data of the image is stored in the **__TEXT** section of the executable, which is encrypted by Apple when distributed through the App Store. If that `__TEXT` section sounded alien, you'll learn about it in Chapter 18: "Hello, Mach-O".

First, you'll explore hooking into a common C function. Once you've mastered the concepts, you'll execute a private Swift method that's unavailable to you at development time thanks to the Swift compiler. Using `dlopen` and `dlsym`, you'll be able to call and execute this private method inside a framework with zero modifications to the framework's code.

Now that you've got more theory than you've ever wanted in an introduction, it's finally time to get started.

Easy mode: hooking C functions

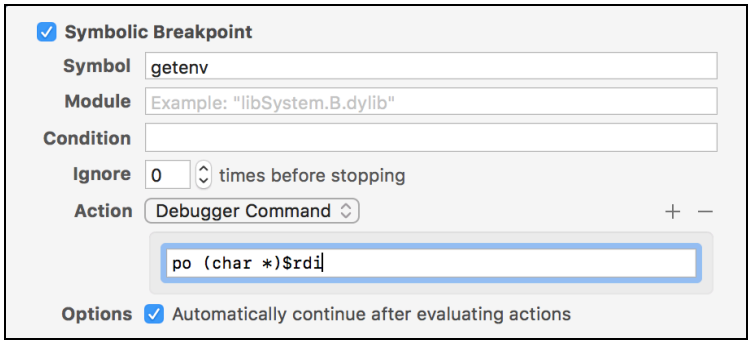
When learning how to use the `dlopen` and `dlsym` functions, you'll be going after the `getenv` C function. This simple C function takes a `char *` (null terminated string) for input and returns the environment variable for the parameter you supply.

This function is actually called quite a bit when your executable starts up.

Open and launch the **Watermark** project in Xcode. Create a new symbolic breakpoint, putting `getenv` in the **Symbol** section. Next, add a custom action with the following:

```
po (char *)$rdi
```

Now, make sure the execution automatically continues after the breakpoint hits.



Finally, build and run the application on the iPhone XS Simulator, then watch the console. You'll get a slew of output indicating this method is called quite frequently.

It'll look similar to the following:

```
"DYLD_INSERT_LIBRARIES"
"NSZombiesEnabled"
"OBJC_DEBUG_POOL_ALLOCATION"
"MallocStackLogging"
"MallocStackLoggingNoCompact"
"OBJC_DEBUG_MISSING_POOLS"
"LIBDISPATCH_DEBUG_QUEUE_INVERSIONS"
"LIBDISPATCH_CONTINUATION_ALLOCATOR"
... etc ...
```

Note: A far more elegant way to dump all environment variables available to your application is to use the `DYLD_PRINT_ENV`. To set this up, go to **Product\Manage Scheme**, and then add this in the `Environment` variables section. You can simply add the name, **DYLD_PRINT_ENV**, with no value, to dump out all environment variables at runtime.

However, an important point to note is all these calls to `getenv` are happening before your executable has even started. You can verify this by putting a breakpoint on `getenv` and looking at the stack trace. Notice `main` is nowhere in sight. This means you'll not be able to alter these function calls until your code can get executed.

Since C doesn't use dynamic dispatch, hooking a function requires you to intercept the function before it's loaded. On the plus side, C functions are relatively easy to grab. All you need is the name of the C function without any parameters along with the name of the dynamic framework in which the C function is implemented.

However, since C is all-powerful and used pretty much everywhere, there are different tactics of varying complexity you can explore to hook a C function. If you want to hook a C function inside your own executable, that's not a lot of work. However, if you want to hook a function called before your code (main executable or frameworks) is loaded in by `dyld`, the complexity definitely goes up a notch.

As soon as your executable executes `main`, it's already imported all the dynamic frameworks specified in the load commands, as you learned in the previous chapter. The dynamic linker will recursively load frameworks in a depth-first manner. If you were to call an external framework, it can be lazily loaded or immediately loaded upon module load by `dyld`. Typically, most external functions are lazily loaded unless you specify special linker flags.

With lazily loaded functions, the first time the function is called, a flurry of activity occurs as `dyld` finds the module and location responsible for the function. This value is then put into a specific `section` in memory (`__DATA.__la_symbol_ptr`, but we'll talk about that later). Once the external function is resolved, all future calls to that function will not need to be resolved by `dyld`.

This means if you want to have the function hooked before your application starts up, you'll need to create a dynamic framework to put the hooking logic in so it'll be available before the `main` function is called. You'll explore this easy case of hooking a C function inside your own executable first.

Back to the `Watermarks` project!

Open **AppDelegate.swift**, and replace `application(_:didFinishLaunchingWithOptions:)` with the following:

```
func application(_ application: UIApplication,
didFinishLaunchingWithOptions launchOptions:
[UIApplication.LaunchOptionsKey : Any]? = nil) -> Bool {
  if let cString = getenv("HOME") {
    let homeEnv = String(cString: cString)
    print("HOME env: \(homeEnv)")
  }
  return true
}
```

This creates a call to `getenv` to get the `HOME` environment variable.

Next, remove the symbolic `getenv` breakpoint you previously created and build and run the application.

The console output will look similar to the following:

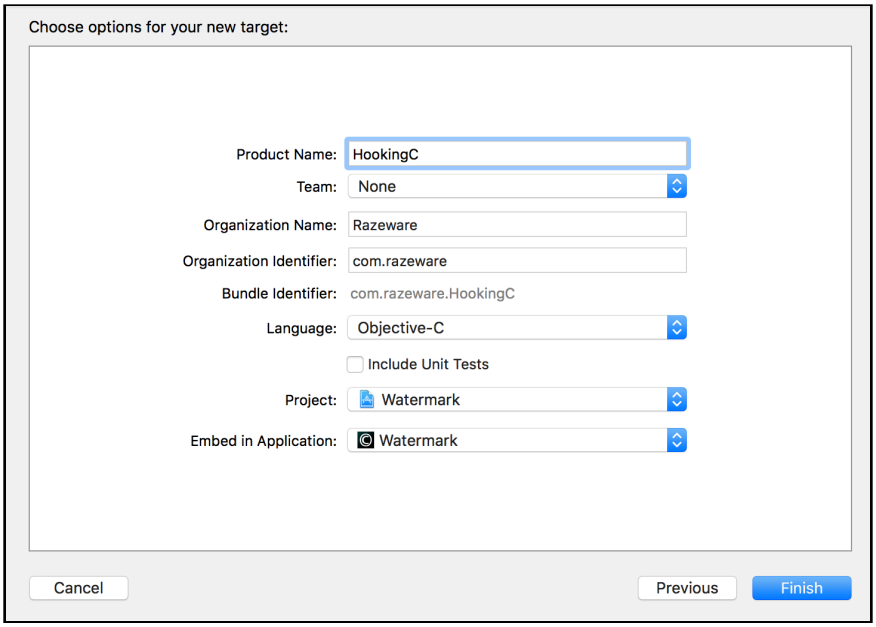
```
HOME env: /Users/derekselander/Library/Developer/CoreSimulator/Devices/
2B9F4587-F75E-4184-861E-C2CAE8F6A1D9/data/Containers/Data/Application/
D7289D91-D73F-47CE-9FAC-E9EED14219E2
```

This is the `HOME` environment variable set for the Simulator you're running on.

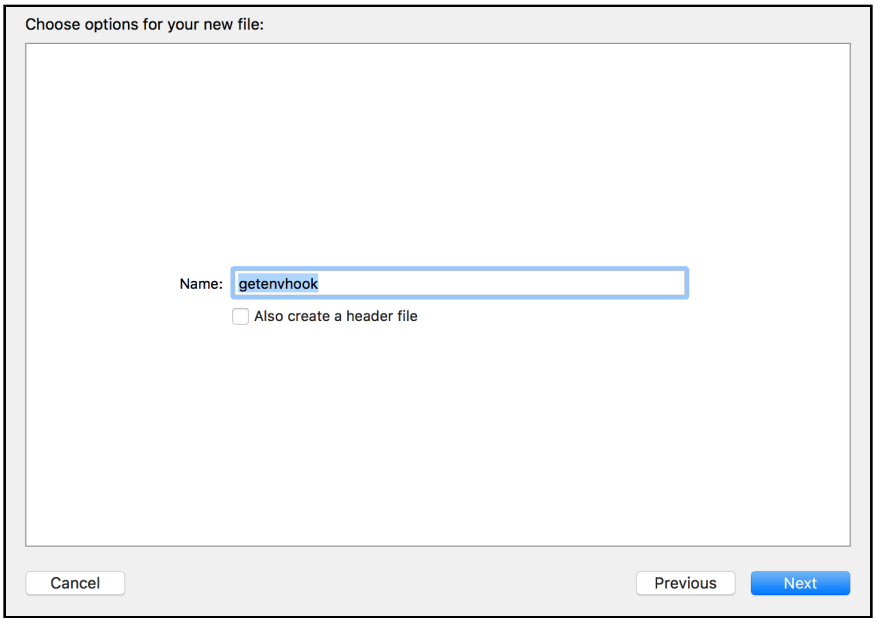
Say you wanted to hook the `getenv` function to act completely normally, but return something different to the output above if and only if `HOME` is the parameter.

As mentioned earlier, you'll need to create a framework that's relied upon by the `Watermark` executable to grab that address of `getenv` and change it before it's resolved in the main executable.

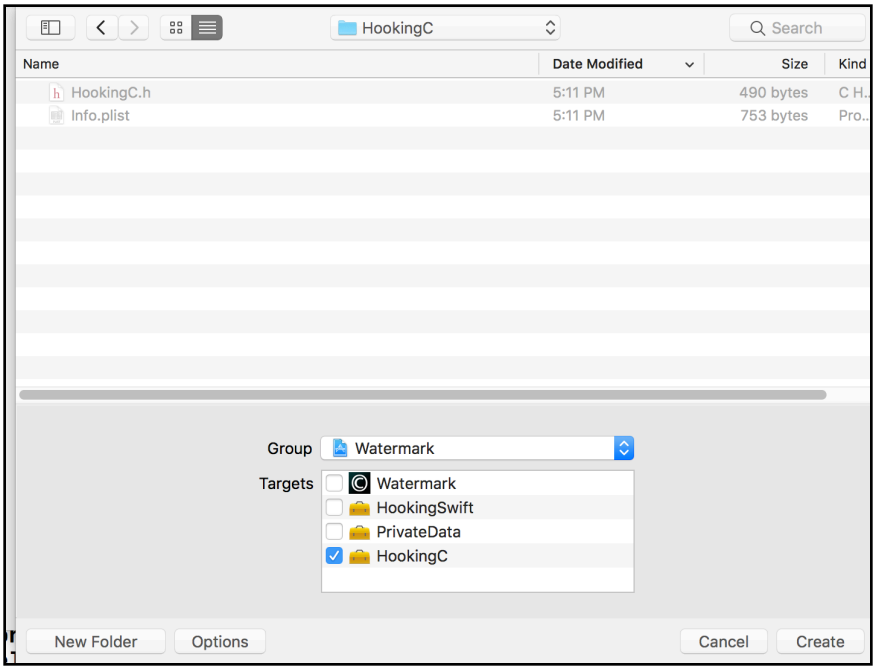
In Xcode, navigate to **File ▸ New ▸ Target** and select **Cocoa Touch Framework**. Choose **HookingC** as the product name, and set the language to **Objective-C**.



Once this new framework is created, create a new C file. In Xcode, select **File\New\File**, then select **C file**. Name this file **getenvhook**. Uncheck the checkbox for **Also create a header file**. Save the file with the rest of the project.



Make sure this file belongs to the **HookingC** framework that you've just created, and **not Watermark**.



OK... you're finally about to write some code... I swear.

Open **getenvhook.c** and replace its contents with the following:

```
#import <dlfcn.h>
#import <assert.h>
#import <stdio.h>
#import <dispatch/dispatch.h>
#import <string.h>
```

- `dlfcn.h` will be responsible for two very interesting functions: `dlopen` and `dlsym`.
- `assert.h` will test the library containing the real `getenv` is correctly loaded.
- `stdio.h` will be used temporarily for a C `printf` call.
- `dispatch.h` will be used to to properly set up the logic for GCD's `dispatch_once` function.
- `string.h` will be used for the `strcmp` function, which compares two C strings.

Next, redeclare the `getenv` function with the hard-coded stub shown below:

```
char * getenv(const char *name) {
  return "YAY!";
}
```

Finally, build and run your application to see what happens. You'll get the following output:

```
HOME env: YAY!
```

Awesome! You were able to successfully replace this method with your own function. However, this isn't quite what you want. You want to call the original `getenv` function and augment the return value if `"HOME"` is supplied as input.

What would happen if you tried to call the original `getenv` function inside your `getenv` function? Try it out and see what happens. Add some temporary code so the `getenv` looks like the following:

```
char * getenv(const char *name) {
  return getenv(name);
  return "YAY!";
}
```

Your program will… sort of… run and then eventually crash. This is because you've just created a stack overflow. All references to the previously linked `getenv` have disappeared now that you've created your own `getenv` fuction.

Undo that previous line of code. That idea won't work. You're going to need a different tactic to grab the original `getenv` function.

First things first though, you need to figure out which library holds the `getenv` function. Make sure that problematic line of code is removed, and build and run the application again. Pause execution and bring up the LLDB console. Once the console pops up, enter the following:

```
(lldb) image lookup -s getenv
```

You'll get output looks similar to the following:

```
1 symbols match 'getenv' in /Users/derekselander/Library/Developer/Xcode/
DerivedData/Watermark-frqludlofnmrzcbjnkmuhgeuogmp/Build/Products/Debug-
iphonesimulator/Watermark.app/Frameworks/HookingC.framework/HookingC:
        Address: HookingC[0x0000000000000f60] (HookingC.__TEXT.__text +
0)
        Summary: HookingC`getenv at getenvhook.c:16
1 symbols match 'getenv' in /Applications/Xcode.app/Contents/Developer/
Platforms/iPhoneSimulator.platform/Developer/SDKs/iPhoneSimulator.sdk//
usr/lib/system/libsystem_c.dylib:
        Address: libsystem_c.dylib[0x000000000005f1c4]
(libsystem_c.dylib.__TEXT.__text + 385956)
        Summary: libsystem_c.dylib`getenv
```

You'll get two hits. One of them will be the `getenv` function you created yourself. More importantly, you'll get the location of the `getenv` function you actually care about. It looks like this function is located in **libsystem_c.dylib**, and its full path is at `/usr/lib/system/libsystem_c.dylib`. Remember, the simulator prepends that big long path to these directories, but the dynamic linker is smart enough to search in the correct areas. Everything after `iPhoneSimulator.sdk` is where this framework is actually stored on a real iOS device.

Now you know exactly where this function is loaded, it's time to whip out the first of the amazing "dl" duo, `dlopen`. Its function signature looks like the following:

```
extern void * dlopen(const char * __path, int __mode);
```

`dlopen` expects a fullpath in the form of a `char *` and a second parameter, which is a mode expressed as an integer that determines how `dlopen` should load the module. If successful, `dlopen` returns an opaque handle (a `void *`) ,or `NULL` if it fails.

After `dlopen` (hopefully) returns a reference to the module, you'll use `dlsym` to get a reference to the `getenv` function. `dlsym` has the following function signature:

```
extern void * dlsym(void * __handle, const char * __symbol);
```

`dlsym` expects to take the reference generated by `dlopen` as the first parameter and the name of the function as the second parameter. If everything goes well, `dlsym` will return the function address for the symbol specified in the second parameter or `NULL` if it failed.

Replace your `getenv` function with the following:

```
char * getenv(const char *name) {
  void *handle = dlopen("/usr/lib/system/libsystem_c.dylib",
                        RTLD_NOW);
  assert(handle);
  void *real_getenv = dlsym(handle, "getenv");
  printf("Real getenv: %p\nFake getenv: %p\n",
         real_getenv,
         getenv);
  return "YAY!";
}
```

You used the `RTLD_NOW` mode of `dlopen` to say, "Hey, don't wait or do any cute lazy loading stuff. Open this module right now." After making sure the handle is not `NULL` through a C `assert`, you call `dlsym` to get a handle on the "real" `getenv`.

Build and run the application. You'll get output similar to the following:

```
Real getenv: 0x10d2451c4
Fake getenv: 0x10a8f7de0
2016-12-19 16:51:30.650 Watermark[1035:19708] HOME env: YAY!
```

Your function pointers will be different than my output, but take note of the difference in address between the real and fake `getenv`.

You're starting to see how you'll go about this. However, you'll need to make a few touch-ups to the above code first. For example, you can cast function pointers to the exact type of function you expect to use. Right now, the `real_getenv` function pointer is `void *`, meaning it could be anything. You already know the function signature of `getenv`, so you can simply cast it to that.

Replace your `getenv` function one last time with the following:

```
char * getenv(const char *name) {
  static void *handle;      // 1
  static char * (*real_getenv)(const char *); // 2

  static dispatch_once_t onceToken;
  dispatch_once(&onceToken, ^{  // 3
    handle = dlopen("/usr/lib/system/libsystem_c.dylib",
                    RTLD_NOW);
    assert(handle);
    real_getenv = dlsym(handle, "getenv");
  });

  if (strcmp(name, "HOME") == 0) { // 4
    return "/WOOT";
  }

  return real_getenv(name); // 5
}
```

You might not be used to this amount of C code, so let's break it down:

1. This creates a static variable named `handle`. It's static so this variable will survive the scope of the function. That is, this variable will not be erased when the function exits, but you'll only be able to access it inside the `getenv` function.

2. You're doing the same thing here as you declare the `real_getenv` variable as static, but you've made other changes to the `real_getenv` function pointer. You've cast this function pointer to correctly match the signature of `getenv`. This will allow you to call the real `getenv` function through the `real_getenv` variable. Cool, right?

3. You're using GCD's `dispatch_once` because you really only need to call the setup once. This nicely complements the `static` variables you declared a couple lines above. You don't want to be doing the lookup logic every time your augmented `getenv` runs!

4. You're using C's `strcmp` to see if you're querying the `"HOME"` environment variable. If it's true, you're simply returning `"/WOOT"` to show yourself that you can change around this value. Essentially, you're overriding what the `getenv` function returns.

5. If `"HOME"` is not supplied as an input parameter, then just fall back on the default `getenv`.

Open **AppDelegate.swift**, and replace `application(_:didFinishLaunchingWithOptions:)` with the following:

```
func application(_ application: UIApplication,
                 didFinishLaunchingWithOptions launchOptions:
[UIApplication.LaunchOptionsKey : Any]? = nil) -> Bool {
  if let cString = getenv("HOME") {
    let homeEnv = String(cString: cString)
    print("HOME env: \(homeEnv)")
  }

  if let cString = getenv("PATH") {
    let homeEnv = String(cString: cString)
    print("PATH env: \(homeEnv)")
  }
  return true
}
```

Build and run the application. Provided everything went well, you'll get output similar to the following:

```
HOME env: /WOOT
PATH env: /Applications/Xcode.app/Contents/Developer/Platforms/
iPhoneSimulator.platform/Developer/SDKs/iPhoneSimulator.sdk/usr/bin:/
Applications/Xcode.app/Contents/Developer/Platforms/
iPhoneSimulator.platform/Developer/SDKs/iPhoneSimulator.sdk/bin:/
Applications/Xcode.app/Contents/Developer/Platforms/
iPhoneSimulator.platform/Developer/SDKs/iPhoneSimulator.sdk/usr/sbin:/
Applications/Xcode.app/Contents/Developer/Platforms/
iPhoneSimulator.platform/Developer/SDKs/iPhoneSimulator.sdk/sbin:/
Applications/Xcode.app/Contents/Developer/Platforms/
iPhoneSimulator.platform/Developer/SDKs/iPhoneSimulator.sdk/usr/local/bin
```

As you can see, your hooked `getenv` augmented the `HOME` environment variable, but defaulted to the normal `getenv` for `PATH`.

Although annoying, it's worth driving this point home yet again. If you called a `UIKit` method, and `UIKit` calls `getenv`, your augmented `getenv` function will not get called during this time because the `getenv`'s address had already been resolved when `UIKit`'s code loaded.

In order to change around `UIKit`'s call to `getenv`, you would need knowledge of the indirect symbol table and to modify the `getenv` address stored in the `__DATA.__la_symbol_ptr` section of the `UIKit` module. This is something you'll learn about in a later chapter.

Hard mode: hooking Swift methods

Going after Swift code that isn't dynamic is a lot like going after C functions. However, there are a couple of complications with this approach that make it a bit harder to hook into Swift methods.

First off, Swift often uses classes or structs in typical development. This is a unique challenge because `dlsym` will only give you a C function. You'll need to augment this function so the Swift method can reference `self` if you're grabbing an instance method, or reference the class if you're calling a class method. When accessing a method that belongs to a class, the assembly will often reference offsets of `self` or the class when performing the method. Since `dlysm` will grab you a C-type function, you'll need to creatively utilize your knowledge of assembly, parameters and registers to turn that C function into a Swift method.

The second issue you need to worry about is that Swift mangles the names of its methods. The happy, pretty name you see in your code is actually a scary long name in the module's symbol table. You'll need to find this method's correct mangled name in order to reference the Swift method through `dlysm`.

As you know, this project produces and displays a watermarked image. Here's the challenge for you: using only code, display the original image in the `UIImageView`. You're not allowed to use LLDB to execute the command yourself, nor are you allowed to modify any contents in memory once the program is running.

Are you up for this challenge? Don't worry, I'll show you how it's done!

First, open **AppDelegate.swift** and remove all the printing logic found inside `application(_:didFinishLaunchingWithOptions:)`. Next, open **CopyrightImageGenerator.swift**.

Inside this class is a `private` computed property containing the `originalImage`. In addition, there's a public computed property containing the `watermarkedImage`. It's this method that calls the `originalImage` and superimposes the watermark. It's up to you to figure out a way to call this `originalImage` method, without changing the `HookingSwift` dynamic library at all.

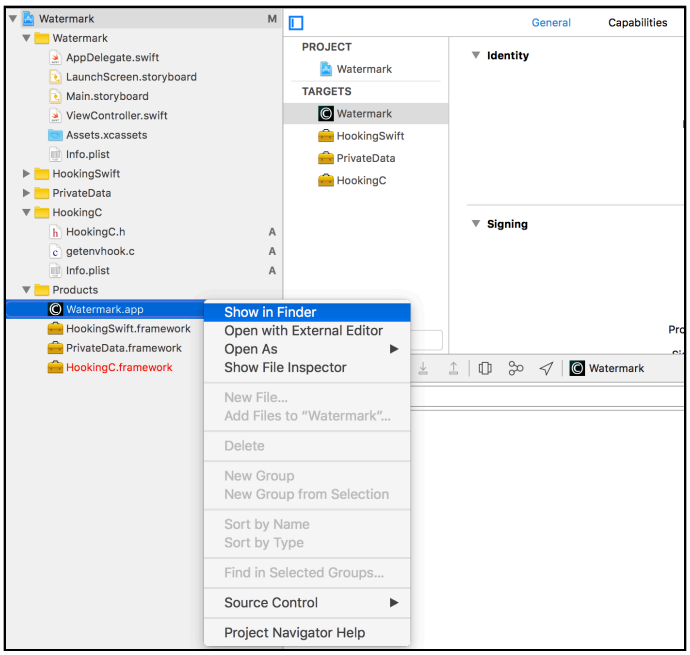
Open **ViewController.swift** and add the following code to the end of `viewDidLoad()`:

```
if let handle = dlopen("", RTLD_NOW) {}
```

You're using Swift this time, but you'll use the same `dlopen` & `dlsym` trick you saw earlier. You now need to get the correct location of the `HookingSwift` framework. The nice thing about `dlopen` is you can supply relative paths instead of absolute paths.

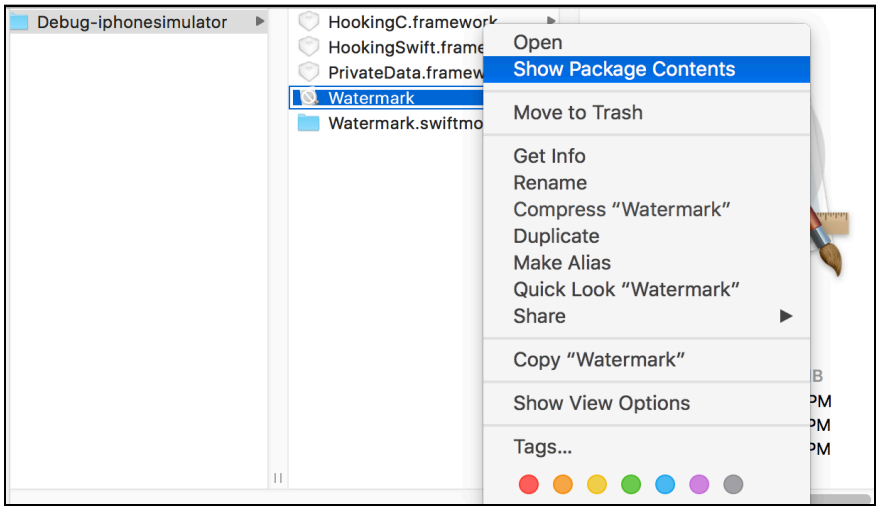
Time to find where that framework is relative to the `Watermark` executable.

In Xcode, make sure the **Project Navigator** is visible (through **Cmd + 1**). Next, open the **Products** directory and right-click the `Watermark.app`. Next, select **Show in Finder**.



Once the Finder window pops up, right click the `Watermark` bundle and select **Show Package Contents**.

It's in this directory the actual `Watermark` executable is located, so you simply need to find the location of the `HookingSwift` framework's executable relative to this `Watermark` executable.



Next, select the Frameworks directory. Finally select the **HookingSwift.framework**. Within this directory, you'll come across the `HookingSwift` binary.

This means you've found the relative path you can supply to `dlopen`. Modify the `dlopen` function call you just added so it looks like the following:

```
if let handle = dlopen("./Frameworks/HookingSwift.framework/
HookingSwift", RTLD_NOW) {
}
```

Now to the hard part. You want to grab the name of the method responsible for the `originalImage` property inside the `CopyrightImageGenerator` class. By now, you know you can use the `image lookup` LLDB function to search for method name compiled into an executable.

Since you know `originalImage` is implemented in Swift, use a "Swift style" type of search with the `image lookup` command. Make sure the app is running, then type the following into LLDB:

```
(lldb) image lookup -rn HookingSwift.*originalImage
```

You'll get output similar to the following:

```
1 match found in /Users/derekselander/Library/Developer/Xcode/
DerivedData/Watermark-gbmzjibibkpgfjefjidpgkfzlakw/Build/Products/Debug-
iphonesimulator/Watermark.app/Frameworks/HookingSwift.framework/
HookingSwift:
        Address: HookingSwift[0x0000000000001550]
(HookingSwift.__TEXT.__text + 368)
```

```
        Summary: HookingSwift`HookingSwift.CopyrightImageGenerator.
(originalImage in _71AD57F3ABD678B113CF3AD05D01FF41).getter :
Swift.Optional<__C.UIImage> at CopyrightImageGenerator.swift:36
```

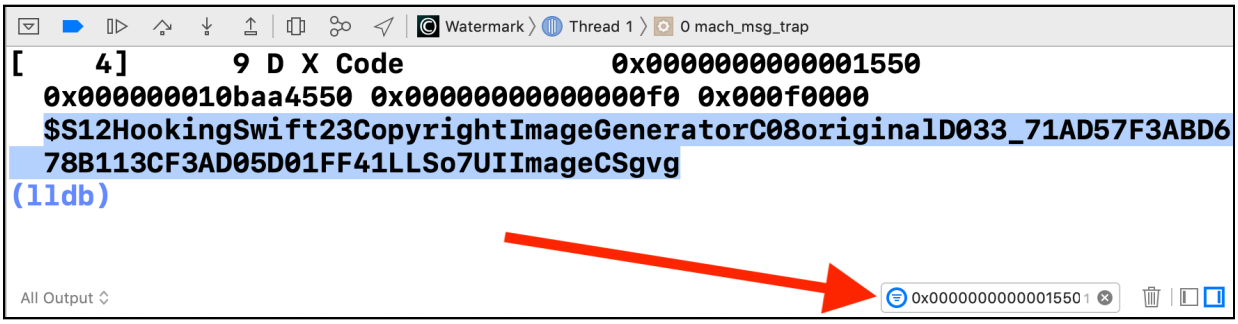
In the output, search for the line containing `Address: HookingSwift[0x0000000000001550]`. This is where this method is implemented inside the `HookingSwift` framework. This will likely be a different address for you.

For this particular example, the function is implemented at offset `0x0000000000001550` inside the `HookingSwift` framework. Copy this address and enter the following command into LLDB:

```
(lldb) image dump symtab -m HookingSwift
```

This dumps the symbol table of the `HookingSwift` framework. In addition to dumping the symbol table, you've told LLDB to show the mangled names of the Swift functions. There will be quite a few symbols that pop up in the display. Paste that address you copied into the LLDB search bar so the scary amount of output becomes managable.

You'll get an address that matches the address you copied:



Here's the line that interests you.

```
[    4]      9 D X Code            0x0000000000001550 0x000000010baa4550
0x00000000000000f0 0x000f0000
$S12HookingSwift23CopyrightImageGeneratorC08originalD033_71AD57F3ABD678B1
13CF3AD05D01FF41LLSo7UIImageCSgvg
```

Yep, that huge angry alphanumeric chunk at the end is the Swift mangled function name. It's this monstrosity you'll stick into `dlsym` to grab the address of the `originalImage` getter method.

Open **ViewController.swift** and add the following code inside the `if let` you just added:

```
let sym = dlsym(handle,
"$S12HookingSwift23CopyrightImageGeneratorC08originalD033_71AD57F3ABD678B
113CF3AD05D01FF41LLSo7UIImageCSgvg")!
print("\(sym)")
```

> **Note**: Until Swift stops playing spin the bottle with its ABI naming, these symbol names could (and have!) change from version to version, meaning the mangled function could be different for you.

You've opted for an implicitly unwrapped optional since you want the application to crash if you got the wrong symbol name.

Build and run the application. If everything worked out, you'll get a memory address at the tail end of the console output (yours will likely be different):

```
0x0000000103105770
```

This address is the location to `CopyrightImageGeneratorg`'s `originalImage` method that `dlsym` provided. You can verify this by creating a breakpoint on this address in LLDB:

```
(lldb) b 0x0000000103105770
```

LLDB creates a breakpoint on the following function:

```
Breakpoint 1: where = HookingSwift`HookingSwift.CopyrightImageGenerator.
(originalImage in _71AD57F3ABD678B113CF3AD05D01FF41).getter :
Swift.Optional<__ObjC.UIImage> at CopyrightImageGenerator.swift:35,
address = 0x0000000103105770
```

Great! You can bring up the address of this function at runtime, but how do you go about calling it? Thankfully, you can use the `typealias` Swift keyword to cast functions signatures.

Open **ViewController.swift**, and add the following directly under the `print` call you just added:

```
typealias privateMethodAlias = @convention(c) (Any) -> UIImage? // 1
let originalImageFunction = unsafeBitCast(sym, to:
privateMethodAlias.self) // 2
let originalImage = originalImageFunction(imageGenerator) // 3
self.imageView.image = originalImage // 4
```

Here's what this does:

1. This declares the type of function that is syntactically equivalent to the Swift function for the `originalImage` property getter. There's something very important to notice here. `privateMethodAlias` is designed so it takes one parameter type of `Any`, but the actual Swift function expects no parameters. Why is this?

   It's due to the fact that by looking at the assembly to this method, the reference to `self` is expected in the RDI register. This means you need to supply the instance of the class as the first parameter into the function to trick this C function into thinking it's a Swift method. If you don't do this, there's a chance the application will crash!

2. Now you've made this new alias, you're casting the `sym` address to this new type and calling it `originalImageFunction`.

3. You're executing the method and supplying the instance of the class as the first and only parameter to the function. This will cause the `RDI` register to be properly set to the instance of the class. It'll return the original image without the watermark.

4. You're assigning the `UIImageView`'s image to the original image without the watermark.

With these new changes in, build and run the application. As expected, the original, watermark-free image will now be displayed in the application.



Congratulations — you've discovered two new amazing functions and how to use them properly. Grabbing the location of code at runtime is a powerful feature that lets you access hidden code the compiler normally blocks from you. In addition, it lets you hook into code so you can perform your own modifications at runtime.

Where to go from here?

You're learning how to play around with dynamic frameworks. The previous chapter showed you how to dynamically load them in LLDB. This chapter showed you how to modify or execute Swift or C code you normally wouldn't be able to. In the next chapter, you're going to play with the Objective-C runtime to dynamically load a framework and use Objective-C's dynamic dispatch to execute classes you don't have the APIs for.

This is one of the most exciting features of reverse engineering — so get prepared, and caffeinated, for your foray into the next chapter!

Chapter 17: Exploring & Method Swizzling Objective-C Frameworks

In the previous two chapters, you've explored dynamic loading as well as how to use the `dlopen` and `dlsym` functions. So long as you knew the name of the function, it didn't matter if the compiler tried to hide a function from you.

You'll cap off this round of dynamic framework exploration by digging into Objective-C frameworks using the Objective-C runtime to hook and execute methods of interest.

For this chapter, you'll go after a series of private `UIKIt` classes that help aid in visual debugging. The chief of these private classes, **UIDebuggingInformationOverlay** was introduced in iOS 9.0 and has received widespread attention in May 2017, thanks to @ryanipete's article http://ryanipete.com/blog/ios/swift/objective-c/uidebugginginformationoverlay/ highlighting these classes and usage.

Unfortunately, as of iOS 11, Apple caught wind of developers accessing this class (likely through the popularity of the above article) and has added several checks to ensure that only internal apps that link to `UIKIt` have access to these private debugging classes. As of iOS 12, Apple has not added additional checks, but the logic has migrated from `UIKIt` to `UIKItCore` (likely) due to the upcoming macOS/iOS integration announced at WWDC 2018. In iOS 12, `UIKIt` doesn't contain *any* code, but simply just links to several frameworks.

You'll explore `UIDebuggingInformationOverlay` and learn why this class fails to work in iOS 11 and above, as well as explore avenues to get around these checks imposed by Apple by writing to specific areas in memory first through LLDB. Then, you'll learn alternative tactics you can use to enable `UIDebuggingInformationOverlay` through Objective-C's method swizzling.

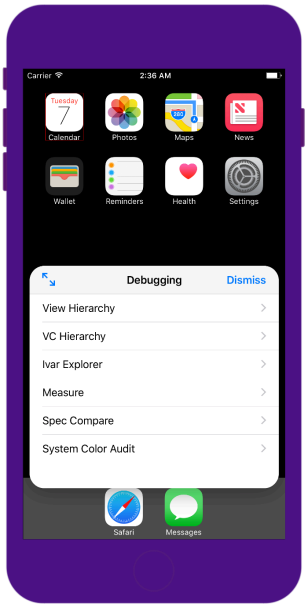
I specifically require you to use iOS 12 for this chapter as Apple can impose new checks on these classes in the future that this chapter doesn't cover.

Between iOS 10 and 12

In iOS 9 & 10, setting up and displaying the overlay was rather trivial. In both these iOS versions, the following LLDB commands were all that was needed:

```
(lldb) po [UIDebuggingInformationOverlay prepareDebuggingOverlay]
(lldb) po [[UIDebuggingInformationOverlay overlay] toggleVisibility]
```

This would produce the following overlay:



If you have an iOS 10 Simulator on your computer, I'd recommend you attach to any iOS process and try the above LLDB commands out so you know what is expected.

Unfortunately, some things changed in iOS 12. Executing the exact same LLDB commands in iOS 12 will produce nothing.

To understand what's happening, you need to explore the overriden methods `UIDebuggingInformationOverlay` contains and wade into the assembly.

Use LLDB to attach to *any* iOS 12 Simulator process, this can be `MobileSafari`, `SpringBoard`, any of the apps you've explored in the previous chapters, or your own work. It doesn't matter if it's your own app or not, as you will be exploring assembly in the `UIKitCore` module.

For this example, I'll launch the **Photos** application in the Simulator. Head on over to Terminal, then type the following:

```
(lldb) lldb -n MobileSlideShow
```

Once you've attached to any iOS Simulator process, use LLDB to search for any overriden methods by the `UIDebuggingInformationOverlay` class.

You can use the `image lookup` LLDB command:

```
(lldb) image lookup -rn UIDebuggingInformationOverlay
```

Or alternatively, use the **methods** command you created in Chapter 15, "Dynamic Frameworks":

```
(lldb) methods UIDebuggingInformationOverlay
```

If you decided to skip that chapter, the following command would be equivalent:

```
(lldb) exp -lobjc -O -- [UIDebuggingInformationOverlay
_shortMethodDescription]
```

Take note of the overridden `init` instance method found in the output of either command.

You'll need to explore what this `init` is doing. You can follow along with LLDB's `disassemble` command, but for visual clarity, I'll use my own custom LLDB disassembler, **dd**, which outputs in color and is available here: https://github.com/DerekSelander/LLDB/blob/master/lldb_commands/disassemble.py.

Here's the `init` method's assembly in iOS 10. If you want to follow along in black & white in LLDB, type:

```
(lldb) disassemble -n "-[UIDebuggingInformationOverlay init]"
```



Again, this is showing the assembly of this method in iOS 10.

Colors (and `dd`'s comments marked in green) make reading x64 assembly soooooooooooo much easier. In pseudo-Objective-C code, this translates to the following:

```
@implementation UIDebuggingInformationOverlay

- (instancetype)init {
  if (self = [super init]) {
    [self _setWindowControlsStatusBarOrientation:NO];
  }
  return self;
}

@end
```

Nice and simple for iOS 10. Let's look at the same method for iOS 12:

```
(lldb) dd 0x10e5a023e
UIKit, -[UIDebuggingInformationOverlay init]
    0    0x10e5a023e <+0>:    push   rbp
    1    0x10e5a023f <+1>:    mov    rbp, rsp
    2    0x10e5a0242 <+4>:    push   r14
    3    0x10e5a0244 <+6>:    push   rbx
    4    0x10e5a0245 <+7>:    sub    rsp, 0x10
    5    0x10e5a0249 <+11>:   mov    rbx, rdi
    6    0x10e5a024c <+14>:   cmp    qword ptr [rip + 0x9fae84], -0x1 UIDebuggingOverlayIsEnabled.__overlayIsEnabled + 7 ; 0x10ef9b0d8 UIKit.__DATA.__bss
   *7    0x10e5a0254 <+22>:   jne    0x10e5a02c0 <+130>
    8    0x10e5a0256 <+24>:   cmp    byte ptr [rip + 0x9fae73], 0x0 mainHandler.onceToken + 7 ; 0x10ef9b0d0 UIKit.__DATA.__bss
   *9    0x10e5a025d <+31>:   je     0x10e5a02a8 <+106>
   10    0x10e5a025f <+33>:   lea    rdi, [rbp - 0x20]
   11    0x10e5a0263 <+37>:   mov    qword ptr [rdi], rbx
   12    0x10e5a0266 <+40>:   mov    rax, qword ptr [rip + 0x99529b] (void *)0x000000010ef7a258: UIDebuggingInformationOverlay ; 0x10ef35508 UIKit.__DATA.__objc_superrefs
   13    0x10e5a026d <+47>:   mov    qword ptr [rdi + 0x8], rax
   14    0x10e5a0271 <+51>:   mov    rsi, qword ptr [rip + 0x945e18] ; "init"
   *15   0x10e5a0278 <+58>:   call   0x10e887126 ; (__TEXT.__stubs) objc_msgSendSuper2
   16    0x10e5a027d <+63>:   mov    rbx, rax
   17    0x10e5a0280 <+66>:   test   rbx, rbx
   *18   0x10e5a0283 <+69>:   je     0x10e5a0297 <+89>
   19    0x10e5a0285 <+71>:   mov    rsi, qword ptr [rip + 0x94a4ac] ; "_setWindowControlsStatusBarOrientation:"
   20    0x10e5a028c <+78>:   xor    edx, edx
   21    0x10e5a028e <+80>:   mov    rdi, rbx
   *22   0x10e5a0291 <+83>:   call   qword ptr [rip + 0x5d2351] (void *)0x0000000116b8c940: objc_msgSend ; 0x10eb725e8 UIKit.__DATA.__got
   23    0x10e5a0297 <+89>:   mov    rdi, rbx
   *24   0x10e5a029a <+92>:   call   qword ptr [rip + 0x5d2358] (void *)0x0000000116b89c50: objc_retain ; 0x10eb725f8 UIKit.__DATA.__got
   25    0x10e5a02a0 <+98>:   mov    rbx, rax
   26    0x10e5a02a3 <+101>:  mov    r14, rbx
   *27   0x10e5a02a6 <+104>:  jmp    0x10e5a02ab <+109>
   28    0x10e5a02a8 <+106>:  xor    r14d, r14d
   29    0x10e5a02ab <+109>:  mov    rdi, rbx
   *30   0x10e5a02ae <+112>:  call   qword ptr [rip + 0x5d233c] (void *)0x0000000116b89cc0: objc_release ; 0x10eb725f0 UIKit.__DATA.__got
   31    0x10e5a02b4 <+118>:  mov    rax, r14
   32    0x10e5a02b7 <+121>:  add    rsp, 0x10
   33    0x10e5a02bb <+125>:  pop    rbx
   34    0x10e5a02bc <+126>:  pop    r14
   35    0x10e5a02be <+128>:  pop    rbp
   *36   0x10e5a02bf <+129>:  ret
   37    0x10e5a02c0 <+130>:  lea    rdi, [rip + 0x9fae11] UIDebuggingOverlayIsEnabled.onceToken ; 0x10ef9b0d8 UIKit.__DATA.__bss
   38    0x10e5a02c7 <+137>:  lea    rsi, [rip + 0x61c002] __block_literal_global.105 ; 0x10ebbc2d0 UIKit.__DATA.__const
   *39   0x10e5a02ce <+144>:  call   0x10e886df6 ; (__TEXT.__stubs) dispatch_once
   *40   0x10e5a02d3 <+149>:  jmp    0x10e5a0256 <+24>
```

This roughly translates to the following:

```
@implementation UIDebuggingInformationOverlay

- (instancetype)init {
  static BOOL overlayEnabled = NO;
  static dispatch_once_t onceToken;
  dispatch_once(&onceToken, ^{
    overlayEnabled = UIDebuggingOverlayIsEnabled();
  });
  if (!overlayEnabled) {
    return nil;
  }

  if (self = [super init]) {
```

```
    [self _setWindowControlsStatusBarOrientation:NO];
  }
  return self;
}

@end
```

There are checks added in iOS 11 and above thanks to **UIDebuggingOverlayIsEnabled()** to return `nil` if this code is not an internal Apple device.

You can verify these disappointing precautions yourself by typing the following in LLDB on a iOS 12 Simulator:

```
(lldb) po [UIDebuggingInformationOverlay new]
```

This is a shorthand way of `alloc/init`'ing an `UIDebuggingInformationOverlay`. You'll get `nil`.

With LLDB, disassemble the first 10 lines of assembly for `–[UIDebuggingInformationOverlay init]`:

```
(lldb) disassemble -n "-[UIDebuggingInformationOverlay init]" -c10
```

Your assembly won't be color coded, but this is a small enough chunk to understand what's going on.

Your output will look similar to:

```
UIKit`-[UIDebuggingInformationOverlay init]:
  0x10d80023e <+0>:  push   rbp
  0x10d80023f <+1>:  mov    rbp, rsp
  0x10d800242 <+4>:  push   r14
  0x10d800244 <+6>:  push   rbx
  0x10d800245 <+7>:  sub    rsp, 0x10
  0x10d800249 <+11>: mov    rbx, rdi
  0x10d80024c <+14>: cmp    qword ptr [rip + 0x9fae84], -0x1
    ; UIDebuggingOverlayIsEnabled.__overlayIsEnabled + 7

  0x10d800254 <+22>: jne    0x10d8002c0               ; <+130>
  0x10d800256 <+24>: cmp    byte ptr [rip + 0x9fae73], 0x0
    ; mainHandler.onceToken + 7

  0x10d80025d <+31>: je     0x10d8002a8               ; <+106>
```

Pay close attention to offset 14 and 22:

```
  0x10d80024c <+14>: cmp    qword ptr [rip + 0x9fae84], -0x1
    ; UIDebuggingOverlayIsEnabled.__overlayIsEnabled + 7

  0x10d800254 <+22>: jne    0x10d8002c0               ; <+130>
```

Thankfully, Apple includes the DWARF debugging information with their frameworks, so we can see what symbols they are using to access certain memory addresses.

Take note of the **UIDebuggingOverlayIsEnabled.__overlayIsEnabled + 7** comment in the disassembly. I actually find it rather annoying that LLDB does this and would consider this a bug. Instead of correctly referencing a symbol in memory, LLDB will reference the previous value in its comments and add a `+ 7`. The value at `UIDebuggingOverlayIsEnabled.__overlayIsEnabled + 7` is what we want, but the comment is not helpful, because it has the name of the wrong symbol in its disassembly. This is why I often choose to use my `dd` command over LLDB's, since I check for this off-by one error and replace it with my own comment.

But regardless of the incorrect name LLDB is choosing in its comments, this address is being compared to `–1` (aka `0xffffffffffffffff` in a 64-bit process) and jumps to a specific address if this address doesn't contain `–1`. Oh… and now that we're on the subject, **`dispatch_once_t`** variables start out as 0 (as they are likely `static`) and get set to `–1` once a **`dispatch_once`** block completes (hint, hint).

Yes, this first check in memory is seeing if code should be executed in a `dispatch_once` block. You want the `dispatch_once` logic to be skipped, so you'll set this value in memory to `–1`.

From the assembly above, you have two options to obtain the memory address of interest:

1. You can combine the `RIP` instruction pointer with the offset to get the load address. In my assembly, I can see this address is located at **`[rip + 0x9fae84]`**. Remember, the `RIP` register will resolve to the next row of assembly since the program counter increments, then executes an instruction.

This means that `[rip + 0x9fae84]` will resolve to `[0x10d800254 + 0x9fae84]` in my case. This will then resolve to **0x000000010e1fb0d8**, the memory address guarding the overlay from being initialized.

2. You can use LLDB's `image lookup` command with the `verbose` and `symbol` option to find the load address for `UIDebuggingOverlayIsEnabled.__overlayIsEnabled`.

```
(lldb) image lookup –vs UIDebuggingOverlayIsEnabled.__overlayIsEnabled
```

From the output, look for the **range** field for the **end address**. Again, this is due to LLDB not giving you the correct symbol. For my process, I got `range = [0x000000010e1fb0d0–0x000000010e1fb0d8)`.

This means the byte of interest for me is located at: `0x000000010e1fb0d8`. If I wanted to know the symbol this address is *actually* referring to, I can type:

```
(lldb) image lookup -a 0x000000010e1fb0d8
```

Which will then output:

```
Address: UIKitCore[0x00000000015b00d8] (UIKitCore.__DATA.__bss + 24824)
Summary: UIKitCore`UIDebuggingOverlayIsEnabled.onceToken
```

This `UIDebuggingOverlayIsEnabled.onceToken` is the correct name of the symbol you want to go after.

Bypassing checks by changing memory

We now know the *exact* bytes where this Boolean check occurs.

Let's first see what value this has:

```
(lldb) x/gx 0x000000010e1fb0d8
```

This will dump out 8 bytes in hex located at 0x000000010e1fb0d8 (your address will be different). If you've executed the `po [UIDebuggingInformationOverlay new]` command earlier, you'll see -1; if you haven't, you'll see 0.

Let's change this. In LLDB type:

```
(lldb) mem write 0x000000010e1fb0d8 0xffffffffffffffff -s 8
```

The `-s` option specifies the amount of bytes to write to. If typing out 16 f's is unappealing to you, there's always alternatives to complete the same task. For example, the following would be equivalent:

```
(lldb) po *(long *)0x000000010e1fb0d0 = -1
```

You can of course verify your work be just examining the memory again.

```
(lldb) x/gx 0x000000010e1fb0d8
```

The output should be `0xffffffffffffffff` now.

Your turn

I just showed you how to knock out the initial check for `UIDebuggingOverlayIsEnabled.onceToken` to make the `dispatch_once` block think it has already run, but there's one more check that will hinder your process.

Re-run the `disassemble` command you typed earlier:

```
(lldb) disassemble -n "-[UIDebuggingInformationOverlay init]" -c10
```

At the very bottom of output are these two lines:

```
0x10d800256 <+24>: cmp    byte ptr [rip + 0x9fae73], 0x0
    ; mainHandler.onceToken + 7
0x10d80025d <+31>: je     0x10d8002a8               ; <+106>
```

This `mainHandler.onceToken` is again, **the wrong symbol**; you care about the symbol immediately following it in memory. I want you to perform the same actions you did on `UIDebuggingOverlayIsEnabled.__overlayIsEnabled`, but instead apply it to the memory address pointed to by the `mainHandler.onceToken` symbol. Once you perform the RIP arithmetic, referencing `mainHandler.onceToken`, you'll realize the correct symbol, **`UIDebuggingOverlayIsEnabled.__overlayIsEnabled`**, is the symbol you are after.

You first need to the find the location of `mainHandler.onceToken` in memory. You can either perform the `RIP` arithmetic from the above assembly or use `image lookup -vs mainHandler.onceToken` to find the end location. Once you found the memory address, write a `-1` value into this memory address.

Verifying your work

Now that you've successfully written a `-1` value to `mainHandler.onceToken`, it's time to check your work to see if any changes you've made have bypassed the initialization checks.

In LLDB type:

```
(lldb) po [UIDebuggingInformationOverlay new]
```

Provided you correctly augmented the memory, you'll be greeted with some more cheery output:

```
<UIDebuggingInformationOverlay: 0x7fb622107860; frame = (0 0; 768 1024);
hidden = YES; gestureRecognizers = <NSArray: 0x60400005aac0>; layer =
<UIWindowLayer: 0x6040000298a0>>
```

And while you're at it, make sure the class method **overlay** returns a valid instance:

```
(lldb) po [UIDebuggingInformationOverlay overlay]
```

If you got `nil` for either of the above LLDB commands, make sure you have augmented the correct addresses in memory. If you're absolutely sure you have augmented the correct addresses and you still get a `nil` return value, make sure you're running either the iOS 12 Simulator as Apple could have added additional checks to prevent this from working in a version since this book was written!

If all goes well, and you have a valid instance, let's put this thing on the screen!

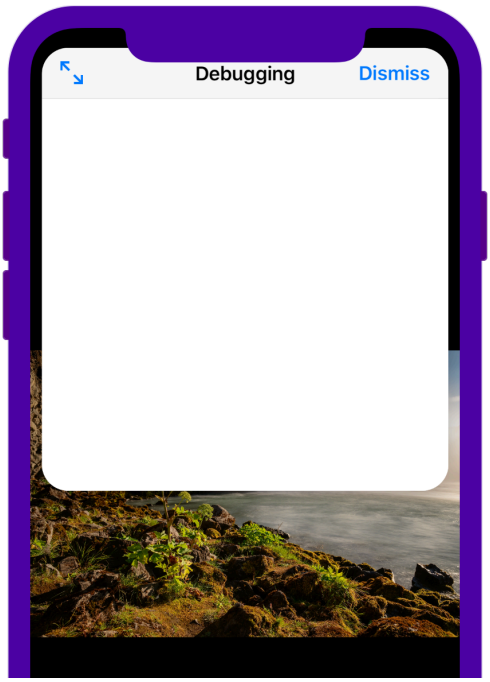
In LLDB, type:

```
(lldb) po [[UIDebuggingInformationOverlay overlay] toggleVisibility]
```

Then resume the process:

```
(lldb) continue
```

Alright... we got something on the screen, but it's blank!?



Sidestepping checks in prepareDebuggingOverlay

The `UIDebuggingInformationOverlay` is blank because we didn't call the class method, **`+[UIDebuggingInformationOverlay prepareDebuggingOverlay]`**

Dumping the assembly for this method, we can see one concerning check immediately:

```
(lldb) dd 0x11385f312
UIKit, +[UIDebuggingInformationOverlay prepareDebuggingOverlay]
    0    0x11385f312 <+0>:    push  rbp
    1    0x11385f313 <+1>:    mov   rbp,  rsp
    2    0x11385f316 <+4>:    push  r15
    3    0x11385f318 <+6>:    push  r14
    4    0x11385f31a <+8>:    push  r13
    5    0x11385f31c <+10>:   push  r12
    6    0x11385f31e <+12>:   push  rbx
    7    0x11385f31f <+13>:   push  rax
   *8    0x11385f320 <+14>:   call  0x1138602bf  ; _UIGetDebuggingOverlayEnabled
    9    0x11385f325 <+19>:   test  al,  al
   *10   0x11385f327 <+21>:   je    0x11385f430 <+286>
    11   0x11385f32d <+27>:   lea   rax,  [rip + 0x9fc19c] UIApp ; 0x11425b4d0 UIKit.__DATA.__common
    12   0x11385f334 <+34>:   mov   rdi,  qword ptr [rax]
    13   0x11385f337 <+37>:   mov   rsi,  qword ptr [rip + 0x94920a]  ; "statusBarWindow"
    14   0x11385f33e <+44>:   mov   r13,  qword ptr [rip + 0x5d22a3] (void *)0x0000000115ba0940: objc_msgSend ; 0x113e315e8 UIKit.__DATA.__got
   *15   0x11385f345 <+51>:   call  r13
    16   0x11385f348 <+54>:   mov   rdi,  rax
   *17   0x11385f34b <+57>:   call  0x113b4614a  ; (__TEXT.__stubs) objc_retainAutoreleasedReturnValue
```

Offsets 14, 19, and 21. Call a function named `_UIGetDebuggingOverlayEnabled` test if `AL` (`RAX`'s single byte cousin) is 0. If yes, jump to the end of this function. The logic in this function is gated by the return value of `_UIGetDebuggingOverlayEnabled`.

Since we are still using LLDB to build a POC, let's set a breakpoint on this function, step out of `_UIGetDebuggingOverlayEnabled`, then augment the value stored in the AL register *before* the check in offset 19 occurs.

Create a breakpoint on `_UIGetDebuggingOverlayEnabled`:

```
(lldb) b _UIGetDebuggingOverlayEnabled
```

LLDB will indicate that it's successfully created a breakpoint on the `_UIGetDebuggingOverlayEnabled` method.

Now, let's execute the `[UIDebuggingInformationOverlay prepareDebuggingOverlay]` method, but have LLDB honor breakpoints. Type the following:

```
(lldb) exp -i0 -O -- [UIDebuggingInformationOverlay
prepareDebuggingOverlay]
```

This uses the `-i` option that determines if LLDB should ignore breakpoints. You're specifying 0 to say that LLDB shouldn't ignore any breakpoints.

Provided all went well, execution will start in the `prepareDebuggingOverlay` method and call out to the `_UIGetDebuggingOverlayEnabled` where execution will stop.

Let's just tell LLDB to resume execution until it steps out of this `_UIGetDebuggingOverlayEnabled` function:

```
(lldb) finish
```

Control flow will finish up in `_UIGetDebuggingOverlayEnabled` and we'll be back in the `prepareDebuggingOverlay` method, right before the test of the AL register on offset 19:

```
UIKit`+[UIDebuggingInformationOverlay prepareDebuggingOverlay]:
    0x11191a312 <+0>:   push   rbp
    0x11191a313 <+1>:   mov    rbp, rsp
    0x11191a316 <+4>:   push   r15
    0x11191a318 <+6>:   push   r14
    0x11191a31a <+8>:   push   r13
    0x11191a31c <+10>:  push   r12
    0x11191a31e <+12>:  push   rbx
    0x11191a31f <+13>:  push   rax
    0x11191a320 <+14>:  call   0x11191b2bf
          ; _UIGetDebuggingOverlayEnabled

->  0x11191a325 <+19>:  test   al, al
    0x11191a327 <+21>:  je     0x11191a430               ; <+286>
    0x11191a32d <+27>:  lea    rax, [rip + 0x9fc19c]     ; UIApp
```

Through LLDB, print out the value in the AL register:

```
(lldb) p/x $al
```

Unless you work at a specific fruit company inside a fancy new “spaceship” campus, you'll likely get `0x00`.

Change this around to `0xff`:

```
(lldb) po $al = 0xff
```

Let's verify this worked by single instruction stepping:

```
(lldb) si
```

This will get you onto the following line:

```
 je     0x11191a430               ; <+286>
```

If `AL` was `0x0` at the time of the `test` assembly instruction, this will move you to offset 286. If `AL` wasn't `0x0` at the time of the `test` instruction, you'll keep on executing without the conditional `jmp` instruction.

Make sure this succeeded by performing one more instruction step.

```
(lldb) si
```

If you're on offset 286, this has failed and you'll need to repeat the process. However, if you find the instruction pointer has not conditionally jumped, then this has worked!

There's nothing more you need to do now, so resume execution in LLDB:

```
(lldb) continue
```

So, what did the logic do exactly in `+[UIDebuggingInformationOverlay prepareDebuggingOverlay]`?

To help ease the visual burden, here is a rough translation of what the `+[UIDebuggingInformationOverlay prepareDebuggingOverlay]` method is doing:

```
+ (void)prepareDebuggingOverlay {
  if (_UIGetDebuggingOverlayEnabled()) {
    id handler = [UIDebuggingInformationOverlayInvokeGestureHandler
mainHandler];
    UITapGestureRecognizer *tapGesture = [[UITapGestureRecognizer alloc]
initWithTarget:handler action:@selector(_handleActivationGesture:)];
    [tapGesture setNumberOfTouchesRequired:2];
    [tapGesture setNumberOfTapsRequired:1];
    [tapGesture setDelegate:handler];

    UIView *statusBarWindow = [UIApp statusBarWindow];
    [statusBarWindow addGestureRecognizer:tapGesture];
  }
}
```

This is interesting: There is logic to handle a two finger tap on UIApp's `statusBarWindow`. Once that happens, a method called `_handleActivationGesture:` will be executed on a `UIDebuggingInformationOverlayInvokeGestureHandler` singleton, `mainHandler`.

That makes you wonder what's the logic in **`-[UIDebuggingInformationOverlayInvokeGestureHandler _handleActivationGesture:]`** is for?

A quick assembly dump using `dd` brings up an interesting area:

```
(lldb) dd 0x11014bf85
UIKit, -[UIDebuggingInformationOverlayInvokeGestureHandler _handleActivationGesture:]
    0    0x11014bf85 <+0>:    push   rbp
    1    0x11014bf86 <+1>:    mov    rbp, rsp
    2    0x11014bf89 <+4>:    push   r15
    3    0x11014bf8b <+6>:    push   r14
    4    0x11014bf8d <+8>:    push   r12
    5    0x11014bf8f <+10>:   push   rbx
    6    0x11014bf90 <+11>:   mov    rbx, rdi
    7    0x11014bf93 <+14>:   mov    rsi, qword ptr [rip + 0x946ffe]  ; "state"
    8    0x11014bf9a <+21>:   mov    rdi, rdx
   *9    0x11014bf9d <+24>:   call   qword ptr [rip + 0x5d2645] (void *)0x00000001173e9940: objc_msgSend ; 0x11071e5e8 UIKit.__DATA.__got
   10    0x11014bfa3 <+30>:   cmp    rax, 0x3
   *11   0x11014bfa7 <+34>:   jne    0x11014c0eb <+358>
   12    0x11014bfad <+40>:   mov    r15, qword ptr [rip + 0x9ac63c] UIDebuggingInformationOverlayInvokeGestureHandler._didCreateTools ; 0x110a
f85f0 UIKit.__DATA.__objc_ivar
   13    0x11014bfb4 <+47>:   cmp    byte ptr [rbx + r15], 0x0
   *14   0x11014bfb9 <+52>:   jne    0x11014c0aa <+293>
   15    0x11014bfbf <+58>:   mov    rdi, qword ptr [rip + 0x9918ea]  ; (__DATA.__objc_classrefs) UIDebuggingInformationHierarchyViewController
   16    0x11014bfc6 <+65>:   mov    r14, qword ptr [rip + 0x94647b]  ; "class"
   17    0x11014bfcd <+72>:   mov    r12, qword ptr [rip + 0x5d2614] (void *)0x00000001173e9940: objc_msgSend ; 0x11071e5e8 UIKit.__DATA.__got
   18    0x11014bfd4 <+79>:   mov    rsi, r14
   *19   0x11014bfd7 <+82>:   call   r12
   20    0x11014bfda <+85>:   lea    rdi, [rip + 0x67b5ff]  ; (__DATA.__cfstring) "View Hierarchy"
```

The `UITapGestureRecognizer` instance passed in by the `RDI` register (which you learned about in Chapter 11, "Assembly Register Calling Convention"), is getting the `state` compared to the value `0x3` (see offset 30). If it is 3, then control continues, while if it's not 3, control jumps towards the end of the function.

A quick lookup in the header file for `UIGestureRecognizer`, tells us the state has the following enum values:

```
typedef NS_ENUM(NSInteger, UIGestureRecognizerState) {
    UIGestureRecognizerStatePossible,
    UIGestureRecognizerStateBegan,
    UIGestureRecognizerStateChanged,
    UIGestureRecognizerStateEnded,
    UIGestureRecognizerStateCancelled,
    UIGestureRecognizerStateFailed,
    UIGestureRecognizerStateRecognized = UIGestureRecognizerStateEnded
};
```

Counting from 0, we can see control will only execute the bulk of the code if the `UITapGestureRecognizer`'s `state` is equal to **UIGestureRecognizerStateEnded**.

So what does this mean exactly? Not only did `UIKIt` developers put restrictions on accessing the `UIDebuggingInformationOverlay` class (which you've already modified in memory), they've also added a "secret" `UITapGestureRecognizer` to the status bar window that executes the setup logic only when you complete a two finger tap on it.

How cool is that?

So, recapping...

Before we try this thing out, let's quickly recap what you did just in case you need to restart fresh:

You found the memory address of **`UIDebuggingOverlayIsEnabled.onceToken`**:

```
(lldb) image lookup -vs UIDebuggingOverlayIsEnabled.onceToken
```

And then set it to `-1` via LLDB's `memory write` or just casting the address to a `long` pointer and setting the value to `-1` like so:

```
(lldb) po *(long *)0x000000010e1fb0d0 = -1
```

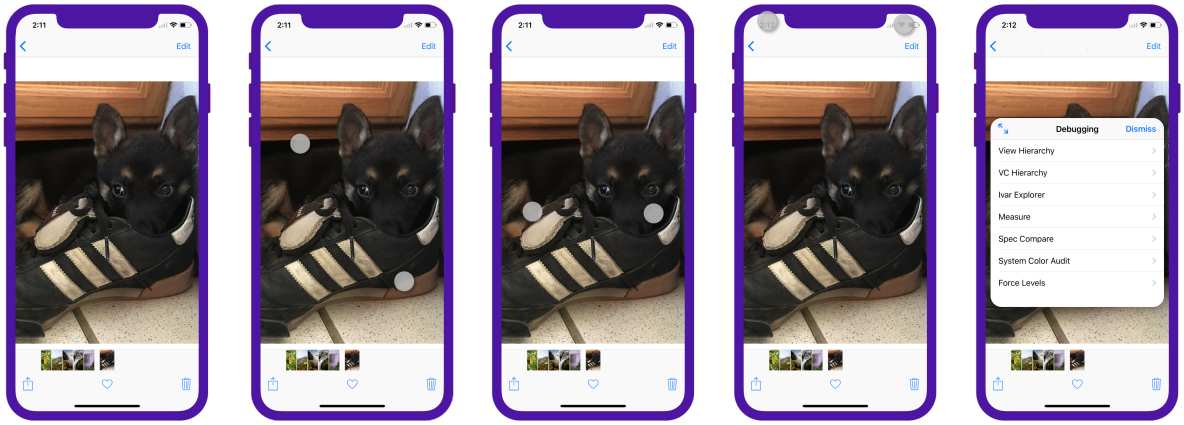
You also performed the same action for **`UIDebuggingOverlayIsEnabled.__overlayIsEnabled`**.

You then created a breakpoint on **`_UIGetDebuggingOverlayEnabled()`**, executed the **`+[UIDebuggingInformationOverlay prepareDebuggingOverlay]`** command and changed the return value that `_UIGetDebuggingOverlayEnabled()` produced so the rest of the method could continue to execute.

This was one of the many ways to bypass Apple's checks to prevent you from using these classes.

Trying this out

Since you're using the Simulator, this means you need to hold down **Option** on the keyboard to simulate two touches. Once you get the two touches parallel, hold down the **Shift** key to drag the tap circles around the screen. Position the tap circles on the status bar of your application, and then click.



You'll be greeted with the fully functional `UIDebuggingInformationOverlay`!

Introducing method swizzling

Reflecting, how long did that take? In addition, we have to manually set this through LLDB everytime `UIKIt` gets loaded into a process. Finding and setting these values in memory can definitely be done through a custom LLDB script, but there's an elegant alternative using Objective-C's **method swizzling**.

But before diving into how, let's talk about the what.

Method swizzling is the process of dynamically changing what an Objective-C method does at runtime. Compiled code in the `__TEXT` section of a binary *can't* be modified (well, it *can* with the proper entitlements that Apple will not give you, but we won't get into that).

However, when executing Objective-C code, we know `objc_msgSend` comes into play thanks to Chapter 11. In case you forgot, `objc_msgSend` will take an instance (or class), a Selector and a variable number of arguments and jump to the location of the function.

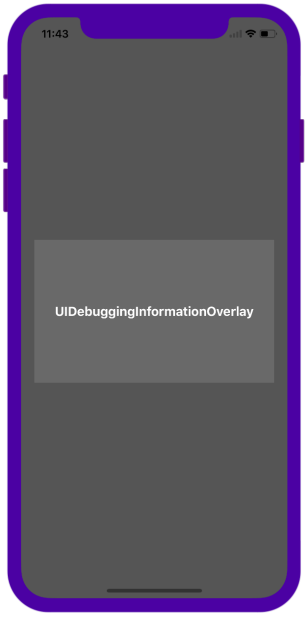
Method swizzling has many uses, but oftentimes people use this tactic to modify a parameter or return value. Alternatively, they can snoop and see when a function is executing code without searching for references in assembly. In fact, Apple even (precariously) uses method swizzling in it's own codebase like KVO!

Since the internet is full of great references on method swizzling, I won't start at square one (but if you want to, I'd say http://nshipster.com/method-swizzling/ has the clearest and cleanest discussion of it).

Instead, we'll start with the basic example, then quickly ramp up to something I haven't seen anyone do with method swizzling: use it to jump into an offset of a method to avoid any unwanted checks!

Finally, onto a sample project

Included in this chapter is an sample project called **Overlay** and it's quite minimal. It only has a `UIButton` smack in the middle that executes the expected logic to display the `UIDebuggingInformationOverlay`.



You'll build an Objective-C `NSObject` category to perform the Objective-C swizzling on the code of interest as soon as the module loads, using the Objective-C-only **load** class method.

Build and run the project. Tap on the lovely `UIButton`. You'll only get some angry output from `stderr` saying:

```
UIDebuggingInformationOverlay 'overlay' method returned nil
```

As you already know, this is because of the short-circuited overriden `init` method for `UIDebuggingInformationOverlay`.

Let's knock out this easy swizzle first; open **NSObject+UIDebuggingInformationOverlayInjector.m**. Jump to Section 1, marked by a **pragma**. In this section, add the following Objective-C class:

```
//***************************************************/
#pragma mark - Section 1 - FakeWindowClass
//***************************************************/

@interface FakeWindowClass : UIWindow
@end
```

```
@implementation FakeWindowClass

- (instancetype)initSwizzled
{
  if (self= [super init]) {
    [self _setWindowControlsStatusBarOrientation:NO];
  }
  return self;
}

@end
```

For this part, you declared an Objective-C class named `FakeWindowClass`, which is a subclass of a `UIWindow`. Unfortunately, this code will not compile since `_setWindowControlsStatusBarOrientation:` is a private method.

Jump up to section 0 and forward declare this private method.

```
//%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%/
#pragma mark - Section 0 - Private Declarations
//%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%/

@interface NSObject()
- (void)_setWindowControlsStatusBarOrientation:(BOOL)orientation;
@end
```

This will quiet the compiler and let the code build. The `UIDebuggingInformationOverlay`'s `init` method has checks to return `nil`. Since the `init` method was rather simple, you just completely sidestepped this logic and reimplemented it yourself and removed all the "bad stuff"!

Now, replace the code for `UIDebuggingInformationOverlay`'s `init` with `FakeWindowClass`'s `initSwizzled` method. Jump down to section 2 in `NSObject`'s `load` method and replace the `load` method with the following:

```
+ (void)load
{
  static dispatch_once_t onceToken;
  dispatch_once(&onceToken, ^{
    Class cls = NSClassFromString(@"UIDebuggingInformationOverlay");
    NSAssert(cls, @"DBG Class is nil?");

    // Swizzle code here

    [FakeWindowClass swizzleOriginalSelector:@selector(init)
                        withSizzledSelector:@selector(initSwizzled)
                                   forClass:cls
                              isClassMethod:NO];
  });
}
```

Rerun and build the Overlay app with this new code. Tap on the `UIButton` to see what happens now that you've replaced the `init` to produce a valid instance.



`UIDebuggingInformationOverlay` now pops up without any content. Almost there!

The final push

You're about to build the final snippet of code for the soon-to-be-replacement method of `prepareDebuggingOverlay`. `prepareDebuggingOverlay` had an initial check at the beginning of the method to see if `_UIGetDebuggingOverlayEnabled()` returned `0x0` or `0x1`. If this method returned `0x0`, then control jumped to the end of the function.

In order to get around this, you'll replicate the same actions you observed in Chapter 13, "Assembly and the Stack" for x86 assembly. That is, you'll "simulate" a **`call`** instruction by pushing a return address onto the stack, but instead of `call`'ing, you'll **`jmp`** into an offset past the `_UIGetDebuggingOverlayEnabled` check. That way, you can perform the function proglogue in your stack frame and directly skip the dreaded check in the beginning of `prepareDebuggingOverlay`.

In **NSObject+UIDebuggingInformationOverlayInjector.m**, Navigate down to **Section 3 - prepareDebuggingOverlay**, and add the following snippet of code:

```
+ (void)prepareDebuggingOverlaySwizzled {
  Class cls = NSClassFromString(@"UIDebuggingInformationOverlay");
  SEL sel = @selector(prepareDebuggingOverlaySwizzled);
  Method m = class_getClassMethod(cls, sel);
  IMP imp =  method_getImplementation(m); // 1

  void (*methodOffset) = (void *)((imp + (long)27)); // 2
```

```
    void *returnAddr = &&RETURNADDRESS; // 3

    // You'll add some assembly here in a sec
    RETURNADDRESS: ;  // 4
}
```

Let's break this crazy witchcraft down:

1. Get the starting address of the original `prepareDebuggingOverlay`. However, this will be swizzled code, so when this code executes, `prepareDebuggingOverlaySwizzled` will actually point to the real, `prepareDebuggingOverlay` starting address.

2. Take the starting address of the original `prepareDebuggingOverlay` (given via the `imp` variable) and offset the value in memory past the `_UIGetDebuggingOverlayEnabled()` check. LLDB is used to figure the exact offset by dumping the assembly and calculating the offset (`disassemble -n "+ [UIDebuggingInformationOverlay prepareDebuggingOverlay]"`). This is *insanely* brittle as any new code or compiler changes from clang will likely break this. I strongly recommend you calculate this yourself in case this changes past iOS 12. You'll look at a more elegant alternative to this implementation after this.

3. Since you are faking a function call, you need an address to return to after this soon-to-be-executed function offset finishes. This is accomplished by getting the address of a declared label. Labels are a not often used feature by normal developers which allow you to `jmp` to different areas of a function. The use of labels in modern programming is considered bad practice as if/for/while loops can accomplish the same thing... but not for this crazy hack.

4. This is the declaration of the label `RETURNADDRESS`. No, you *do* need that semicolon after the label as the C syntax for a label to have a statement immediately following it.

Time to cap this bad boy off with some sweet inline assembly! Right above the label `RETURNADDRESS` declaration, add the following inline assembly:

```
+ (void)prepareDebuggingOverlaySwizzled {
  Class cls = NSClassFromString(@"UIDebuggingInformationOverlay");
  SEL sel = @selector(prepareDebuggingOverlaySwizzled);
  Method m = class_getClassMethod(cls, sel);

  IMP imp =  method_getImplementation(m);
  void (*methodOffset) = (void *)((imp + (long)27));
  void *returnAddr = &&RETURNADDRESS;

  __asm__ __volatile__(     // 1
      "pushq  %0\n\t"       // 2
      "pushq  %%rbp\n\t"    // 3
```

```
        "movq   %%rsp, %%rbp\n\t"
        "pushq  %%r15\n\t"
        "pushq  %%r14\n\t"
        "pushq  %%r13\n\t"
        "pushq  %%r12\n\t"
        "pushq  %%rbx\n\t"
        "pushq  %%rax\n\t"
        "jmp  *%1\n\t"        // 4
        :
        : "r" (returnAddr), "r" (methodOffset)); // 5

    RETURNADDRESS: ;  // 5
}
```

1. Don't be scared, you're about to write x86_64 assembly in AT&T format (Apple's assembler is not a fan of Intel). That `__volatile__` is there to hint to the compiler to not try and optimize this away.

2. You can think of this sort of like C's `printf` where the `%0` will be replaced by the value supplied by the `returnAddr`. In x86, the return address is pushed onto the stack right before entering a function. As you know, `returnAddr` points to an executable address following this assembly. This is how we are faking an actual function call!

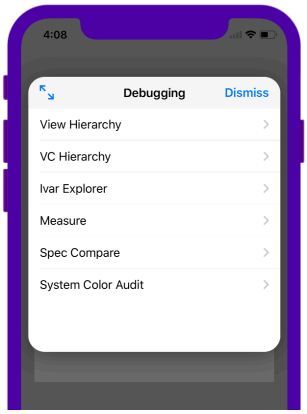
3. The following assembly is copy pasted from the function prologue in the `+[UIDebuggingInformationOverlay prepareDebuggingOverlay]`. This lets us perform the setup of the function, but allows us to skip the dreaded check.

4. Finally we are jumping to offset 27 of the `prepareDebuggingOverlay` after we have set up all the data and stack information we need to not crash. The `jmp *%1` will get resolved to `jmp`'ing to the value stored at `methodOffset`. Finally, what are those "r" strings? I won't get too into the details of inline assembly as I think your head might explode with an information overload (think Scanners), but just know that this is telling the assembler that your assembly can use any register for reading these values.

Jump back up to section 2 where the swizzling is performed in the `+load` method and add the following line of code to the end of the method:

```
[self swizzleOriginalSelector:@selector(prepareDebuggingOverlay)
          withSizzledSelector:@selector(prepareDebuggingOverlaySwizzled)
                     forClass:cls
                isClassMethod:YES];
```

Build and run. Tap on the `UIButton` to execute the required code to setup the `UIDebuggingInformationOverlay` class, then perform the two-finger tap on the status bar.



Omagerd, can you believe that worked?

I am definitely a fan of the hidden status bar dual tap thing, but let's say you wanted to bring this up solely from code. Here's what you can do:

Open **ViewController.swift**. At the top of the file add:

```
import UIKit.UIGestureRecognizerSubclass
```

This will let you set the **state** of a `UIGestureRecognizer` (default headers allow only read-only access to the `state` variable).

Once that's done, augment the code in `overlayButtonTapped(_ sender: Any)` to be the following:

```
@IBAction func overlayButtonTapped(_ sender: Any) {
  guard
    let cls = NSClassFromString("UIDebuggingInformationOverlay") as?
UIWindow.Type else {
      print("UIDebuggingInformationOverlay class doesn't exist!")
      return
  }
  cls.perform(NSSelectorFromString("prepareDebuggingOverlay"))

  let tapGesture = UITapGestureRecognizer()
  tapGesture.state = .ended

  let handlerCls =
NSClassFromString("UIDebuggingInformationOverlayInvokeGestureHandler")
as! NSObject.Type
  let handler = handlerCls
    .perform(NSSelectorFromString("mainHandler"))
    .takeUnretainedValue()
  let _ = handler
```

```
    .perform(NSSelectorFromString("_handleActivationGesture:"),
             with: tapGesture)
}
```

Final build and run. Tap on the button and see what happens.

Boom.

Other implementations

This definitely wasn't the only way to attack the problem. I chose the assembly option since it's fun. But there are cleaner approaches.

For example, the `prepareDebuggingOverlaySwizzled` method could have alernatively done the following which eliminates the need for using inline assembly:

```
#pragma clang diagnostic push
#pragma clang diagnostic ignored "-Wundeclared-selector"

  id handler =
[NSClassFromString(@"UIDebuggingInformationOverlayInvokeGestureHandler")
performSelector:@selector(mainHandler)];

  UITapGestureRecognizer *tapGesture =  [[UITapGestureRecognizer alloc]
initWithTarget:handler action:@selector(_handleActivationGesture:)];
  [tapGesture setDelegate:handler];
  [tapGesture setNumberOfTapsRequired:1];
  [tapGesture setNumberOfTouchesRequired:2];

  [[[UIApplication sharedApplication]
        performSelector:@selector(statusBarWindow)]
                  addGestureRecognizer:tapGesture];

#pragma clang diagnostic pop // end of ignore undeclared selector
```

There's other published ways as well. For example, check out @ian_mcdowell's implementation https://gist.github.com/IMcD23/1fda47126429df43cc989d02c1c5e4a0, which does the whole action in even less code.

Where to go from here?

Crazy chapter, eh? In this chapter, you spelunked into memory and changed `dispatch_once_t` tokens as well as Booleans in memory to build a POC `UIDebuggingInformationOverlay` that's compatible with iOS 12 while getting around Apple's newly introduced checks to prevent you from using this class.

Then you used Objective-C's method swizzling to perform the same actions as well as hook into only a portion of the original method, bypassing several short-circuit checks.

This is why reverse engineering Objective-C is so much fun, because you can hook into methods that are quietly called in private code you don’t have the source for and make changes or monitor what it’s doing.

Still have energy after that brutal chapter? This swizzled code will not work on an ARM64 device. You'll need to look at the assembly and perform an alernative action for that architecture likely through a preprocessor macro.

Oh, and remember how I said that `UIDebuggingInformationOverlay` can totally be made into an LLDB script that's compatible on both the Simulator and an actual device on iOS 12? Well, here it is:

https://github.com/DerekSelander/LLDB/blob/master/lldb_commands/overlaydbg.py

Mic drop.

Enjoy!

Chapter 18: Hello, Mach-O

Mach-O is the file format used for a compiled program running on any of your Apple operating systems. Knowledge of the format is important for both debugging and reverse engineering, since the layout Mach-O defines is applicable to how the executable is stored on disk as well as how the executable is loaded into memory.

Knowing which area of memory an instruction is referencing is useful on the reverse engineering side, but there are a number of useful hidden treasures on the debugging front when exploring Mach-O. For example:

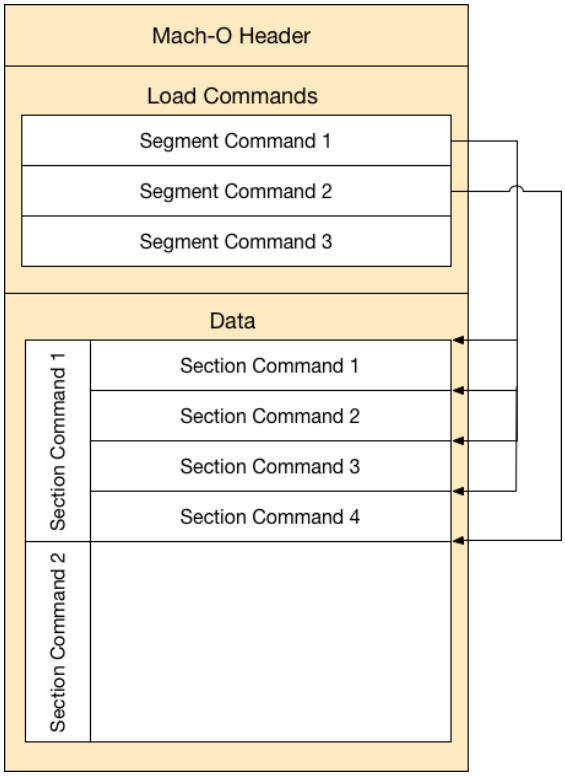
- You can instrospect an external function call at runtime.
- You can quickly find the reference to a singleton's memory address without having to trip a breakpoint.
- You can inspect and modify variables in your own app or other frameworks
- You can perform security audits and make sure no internal, secret messages are being sent out into production in the form of strings or methods.

This chapter introduces the concepts of Mach-O, while the next chapter, **Mach-O Fun** will show the amusing things that are possible with this knowledge. Make sure you have that caffeine on board for this chapter since the theory comes first, followed by the fun in the following chapter.

Terminology

Before diving into the weeds with all the different C `structs` you're about to view, it would be best to take a high level, birds-eye view of the Mach-O layout.

This is the layout of every compiled executable; every main program, every framework, every kernel extension, *everything that's compiled on an Apple platform*.



At the start of every compiled Apple program is the **Mach-O header** that gives information about the CPU this program can run on, the type of executable it is (A framework? A standalone program?) as well as how many **load commands** immediately follow it.

Load commands are instructions on how to load the program and are made up of C `structs`, which vary in size depending on the type of load command.

Some of the load commands provide instructions about how to load **segments**. Think of segments as areas of memory that have a specific type of memory protection. For example, executable code should only have read and execute permissions; it doesn't need write permissions.

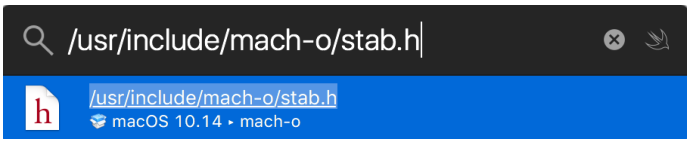
Other parts of the program, such as global variables or singletons, need read and write permissions, but not executable permissions. This means that executable code and the address to global variables will live in separate segments.

Segments can have 0 or more subcomponents called **sections**. These are more finely-grained areas bound by the same memory protections given by their parent segment. Take another look at the above diagram. `Segment Command 1`, points to an offset in the executable that contains four section commands, while `Segment Command 2` points to an offset that contains 0 section commands. Finally, `Segment Command 3` doesn't point to *any* offset in the executable.

It's these sections that can be of profound interest to developers and reverse engineerers since they each serve a unique purpose to the program. For example, there's a specific section to store hard-coded UTF-8 strings, there's a specific section to store references to statically defined variables and so on.

The ultimate goal of these two Mach-O chapters is to show you some interesting load commands in this chapter, and reveal some interesting sections in the next chapter.

In this chapter, you'll be seeing a lot of references to system headers. If you see something like `mach-o/stab.h`, you can view it via the `Open Quickly` menu in Xcode by pressing `⌘ + Shift + O` (the default), then typing in `/usr/include/mach-o/stab.h`.



I'd recommend adding a `/usr/include/` to the search query since Xcode isn't all that smart at times.

If you want to view this header without Xcode, then the physical location will be at:

```
${PATH_TO_XCODE}/Contents/Developer/Platforms/$
{SYSTEM_PLATFORM}.platform/Developer/SDKs/${SYSTEM_PLATFORM}.sdk/usr/
include/mach-o/stab.h
```

Where `${SYSTEM_PLATFORM}` can be `MacOSX`, `iPhoneOS`, `iPhoneSimulator`, `WatchOS`, etc.

Now you've gotten a birds-eye overview, it's time to drop down into the weeds and view all the lovely C `structs`.

The Mach-O header

At the beginning of every compiled Apple executable is a special `struct` that indicates if it's a Mach-O executable. This `struct` can be found in **mach-o/loader.h**. Remember the name of this header file, as it will be referenced quite a bit in this chapter. There are two variants to this `struct`: one for 32-bit operating systems (`mach_header`), and one for 64-bit operating systems (`mach_header_64`). This chapter will talk about 64-bit systems by default, unless otherwise stated.

Let's take a look at the layout of the `struct mach_header_64`.

```
struct mach_header_64 {
  uint32_t  magic;    /* mach magic number identifier */
  cpu_type_t  cputype;  /* cpu specifier */
  cpu_subtype_t cpusubtype; /* machine specifier */
  uint32_t  filetype; /* type of file */
  uint32_t  ncmds;    /* number of load commands */
  uint32_t  sizeofcmds; /* the size of all the load commands */
  uint32_t  flags;    /* flags */
  uint32_t  reserved; /* reserved */
};
```

The first member, **`magic`**, is a hard-coded 32-bit unsigned integer that indicates that this is the beginning of a Mach-O header. What is the value of this `magic` number? A little further down in the `mach-o/loader.h` header, you'll find the following:

```
/* Constant for the magic field of the mach_header_64 (64-bit
architectures) */
#define MH_MAGIC_64 0xfeedfacf /*the 64-bit mach magic number*/
#define MH_CIGAM_64 0xcffaedfe /*NXSwapInt(MH_MAGIC_64)*/
```

This means that every 64-bit Mach-O executable will begin with either `0xfeedfacf`, or `0xcffaedfe` if the byte ordering is swapped. On 32-bit systems, the `magic` value is `0xfeedface`, or `0xcefaedfe` if byte-swapped. It's this value that will let you quickly determine if the file is a Mach-O executable as well as if it's been compiled for a 32-bit or 64-bit architecture.

After the `magic` number are **`cputype`** and **`cpusubtype`**, which indicates on which type of cpu this Mach-O executable is allowed to run. **`filetype`** is useful to know which type of executable you're dealing with. Again, consulting **mach-o/loader.h** shows you the following definitions...

```
#define MH_OBJECT 0x1   /* relocatable object file */
#define MH_EXECUTE  0x2 /* demand paged executable file */
#define MH_FVMLIB 0x3   /* fixed VM shared library file */
#define MH_CORE   0x4   /* core file */
... // there's way more below but ommiting for brevity...
```

So for a main executable (i.e. not a framework), the `filetype` will be `MH_EXECUTE`.

After the `filetype`, the next most interesting aspects of the header are **`ncmds`** and **`sizeofcmds`**. The load commands indicate the attributes and how the executable is loaded into memory.

Time to take break from theory and see this in the wild by examining the raw bytes of an executable's Mach-O header in Terminal.

Mach-O header in grep

Open up a **Terminal** window. I'll pick on the **grep** executable command, but you can pick on any Terminal command that suits your interests. Type the following:

```
xxd -l 32 $(which grep)
```

This command says to dump just the first 32 raw bytes of the fullpath to the location of the `grep` executable. Why 32 bytes? In the `struct mach_header_64` declaration, there are 8 variables, each 4 bytes long.

You'll get something similar to the following:

```
00000000: cffa edfe 0700 0001 0300 0080 0200 0000  ................
00000010: 1300 0000 4007 0000 8500 2000 0000 0000  ....@..... .....
```

Now is a good time to remind yourself that x86_64 bit Intel systems use a **little-endian** architecture. That means that the bytes are reversed.

> **Note**: Even though the x86_64 Intel architecture is little-endian, Apple can store Mach-O information in big-endian or little-endian format, which is partly due to historical reasons dating back to the PPC architecture. iOS doesn't do this, so every iOS file's Mach-O header will be little-endian on disk and in memory. In contrast, the Mach-O header ordering on disk can be found in either format on macOS, but will be little-endian in memory. Later in this section, you'll look at macOS's `CoreFoundation` module, whose Mach-O header is stored in big-endian format. Standards, eh?

Take a closer look at those first 4 bytes from the `xxd` output.

```
cffa edfe
```

This can be split out into individual bytes...

```
cf fa ed fe
```

Then reversed, byte-wise...

```
fe ed fa cf
```

And now, the `MH_MAGIC_64`, a.k.a. the `0xfeedfacf` magic variable, should be evident, indicating this was compiled for a 64-bit system.

Fortunately, the `xxd` Terminal command has a special option for little-endian architectures: the **-e** option. Add the `–e` option to your previous terminal command.

```
xxd –e –l 32 $(which grep)
```

You'll get something similar to the following:

```
00000000: feedfacf 01000007 80000003 00000002  ................
00000010: 00000013 00000740 00200085 00000000  ....@..... .....
```

Let's put all of those values into the `struct mach_header_64`:

```
struct mach_header_64 {
  uint32_t      magic      = 0xfeedfacf
  cpu_type_t    cputype    = 0x01000007
  cpu_subtype_t cpusubtype = 0x80000003
  uint32_t      filetype   = 0x00000002
  uint32_t      ncmds      = 0x00000013
  uint32_t      sizeofcmds = 0x00000740
  uint32_t      flags      = 0x00200085
  uint32_t      reserved   = 0x00000000
};
```

Here you can see the `magic` number of **0xfeedfacf** for the first value. That's a little easier than doing the reversing of the bytes in your head!

After the `0xfeedfacf`, there's a `0x01000007`. To figure this value out, you must consult **mach/machine.h**, which contains the following values:

```
#define CPU_ARCH_ABI64    0x01000000  /* 64 bit ABI */
...
#define CPU_TYPE_X86    ((cpu_type_t) 7)
```

The machine type is `CPU_ARCH_ABI64` ORed together with `CPU_TYPE_X86` producing `0x01000007` in hex (or `16777223` in decimal).

> **Note**: Depending on your computer model, and the version of `grep`, you might receive some different output but the format of the Mach-O will remain the same. Use this as a reference to determine your own unique values.

Likewise, the `cpusubtype` value of `0x80000003` can be determined from the same header file with `CPU_SUBTYPE_LIB64` and `CPU_SUBTYPE_X86_64_ALL` ORed together.

`filetype` has a value of `0x00000002`, or more precisely, `MH_EXECUTE`. There's 19 load commands (`0x00000013` in hex, whose size is `0x00000740`).

The **`flags`** value of `0x00200085` contains a series of options ORed together, but I'll let you jump into **mach-o/loader.h** to figure those out on your own.

If you need a specific homework task, find the significance of the `0x00200000` value in the `flags` variable.

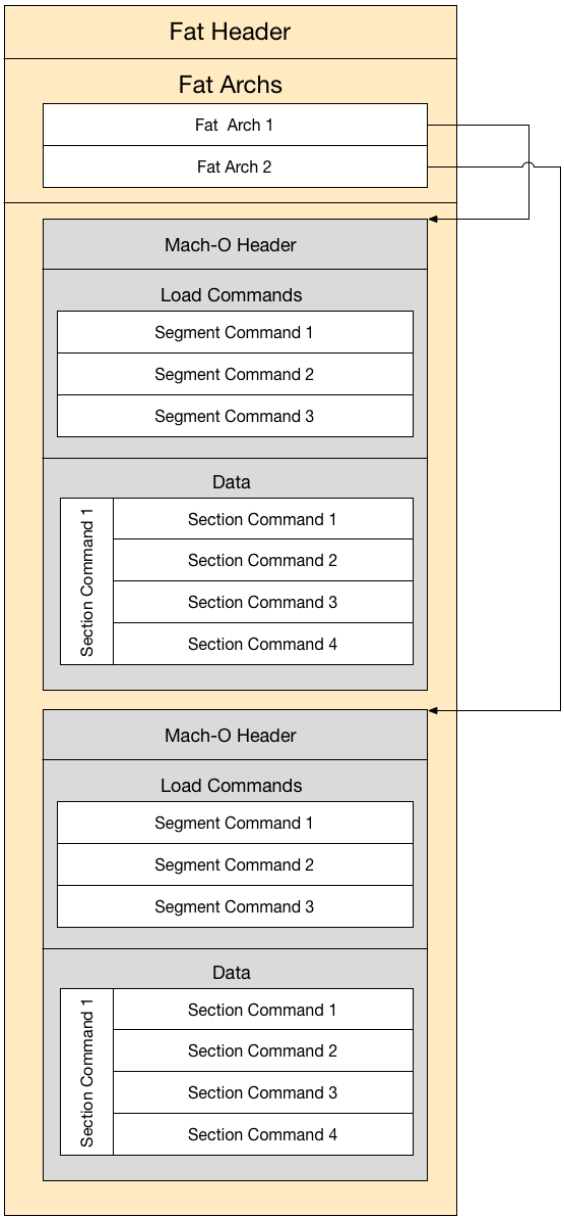
And finally, there is the `reserved` value, which is just a bunch of boring zeros, and means nothing here!

The fat header

Some executables are actually a group of one or more executables "glued" together. For example, many apps compile both a 32-bit and 64-bit executable and place them into a "fat" executable.

This "gluing together" of multiple executables is indicated by a **fat header**, which also has a unique `magic` value differentiating it from a Mach-O header.

Immediately following the fat header are `structs`, which indicate the CPU type and the offset into the file to where the fat header is stored.



Looking at **mach-o/fat.h** gives the following struct:

```
#define FAT_MAGIC 0xcafebabe
#define FAT_CIGAM 0xbebafeca  /* NXSwapLong(FAT_MAGIC) */

struct fat_header {
  uint32_t  magic;    /* FAT_MAGIC or FAT_MAGIC_64 */
  uint32_t  nfat_arch;  /* number of structs that follow */
};

...
#define FAT_MAGIC_64  0xcafebabf
#define FAT_CIGAM_64  0xbfbafeca  /* NXSwapLong(FAT_MAGIC_64) */
```

Although there's a 64-bit equivalent to the fat header, the 32-bit is still widely used in 64-bit systems. The 64-bit fat header is really only used if the offset of the executable slices is greater than 4MB. This is unlike the Mach-O 64-bit variant header you saw in the previous section, which is used only in 64-bit systems.

The fat header contains a number of **fat architecture** structs in the value `nfat_arch` that immediately follow the fat header.

Here's the 64-bit version of the fat architecture:

```
struct fat_arch_64 {
  cpu_type_t  cputype;  /* cpu specifier (int) */
  cpu_subtype_t cpusubtype; /* machine specifier (int) */
  uint64_t  offset;   /* file offset to this object file */
  uint64_t  size;   /* size of this object file */
  uint32_t  align;    /* alignment as a power of 2 */
  uint32_t  reserved; /* reserved */
};
```

And here's the 32-bit version of the fat architecture:

```
struct fat_arch {
  cpu_type_t  cputype;  /* cpu specifier (int) */
  cpu_subtype_t cpusubtype; /* machine specifier (int) */
  uint32_t  offset;   /* file offset to this object file */
  uint32_t  size;   /* size of this object file */
  uint32_t  align;    /* alignment as a power of 2 */
};
```

The `magic` value in the fat header will indicate which of these 32-bit or 64-bit structs to use.

Want to see a real life example of an executable with a fat header? Check out macOS's **CoreFoundation** framework. In Terminal, type this:

```
file /System/Library/Frameworks/CoreFoundation.framework/CoreFoundation
```

You'll see the following:

```
/System/Library/Frameworks/CoreFoundation.framework/CoreFoundation: Mach-
O universal binary with 3 architectures: [x86_64:Mach-O 64-bit
dynamically linked shared library x86_64] [x86_64h]
/System/Library/Frameworks/CoreFoundation.framework/CoreFoundation (for
architecture x86_64): Mach-O 64-bit dynamically linked shared library
x86_64
/System/Library/Frameworks/CoreFoundation.framework/CoreFoundation (for
architecture i386): Mach-O dynamically linked shared library i386
/System/Library/Frameworks/CoreFoundation.framework/CoreFoundation (for
architecture x86_64h):  Mach-O 64-bit dynamically linked shared library
x86_64h
```

This says the `CoreFoundation` consists of three architectures sliced and glued together: **x86_64**, **i386**, **x86_64h**. What's up with the `x86_64h` architecture? It stands for **Haswell**, a x86_64 variant introduced to Macbook Pros in October 2013. On your own time, you can see which architecture is loaded by using LLDB on a program that loads the Core Foundation module.

For example, the following would work when debugging a macOS app that linked to `CoreFoundation`. Open a Terminal window and type the following:

```
lldb $(which plutil)
```

This will open an LLDB session for the `plutil` application, which is a little tool shipped with macOS for manipulating property lists. It just so happens to link Core Foundation, so it's a good one to use for this example.

You'll need to run the application once just so that it actually loads the libraries. Type `run` like so:

```
(lldb) run
Process 946 launched: '/usr/bin/plutil' (x86_64)
No files specified.
plutil: [command_option] [other_options] file...
... etc ...
```

Then inside the LLDB session, type the following:

```
(lldb) image list -h CoreFoundation
[  0] 0x00007fff33cf6000
```

This will dump the load address of the CoreFoundation module. After you've obtained the load address, dump the memory containing the Mach-O header.

```
(lldb) x/8wx 0x00007fff33cf6000
0x7fff33cf6000: 0xfeedfacf 0x01000007 0x00000008 0x00000006
0x7fff33cf6010: 0x00000013 0x00001100 0xc2100085 0x00000000
```

On my computer, `cpusubtype` contains the value `0x00000008`, which equates to the following in `mach/machine.h`:

```
#define CPU_SUBTYPE_X86_64_H    ((cpu_subtype_t)8)  /* Haswell feature
subset */
```

So I can tell that the Haswell, x86_64h slice of `CoreFoundation` was loaded into my process.

Jumping back to the on-disk representation of `CoreFoundation`, dump the raw bytes. Exit out of LLDB and type the following:

```
xxd -l 68 -e /System/Library/Frameworks/CoreFoundation.framework/
CoreFoundation
```

This will produce output similar to:

```
00000000: bebafeca 03000000 07000001 03000000  ................
00000010: 00100000 30767400 0c000000 07000000  .....tv0........
00000020: 03000000 00907400 e0ca6700 0c000000  .....t...g......
00000030: 07000001 08000000 0060dc00 d0e67400  .........`..t..
00000040: 0c000000                             ....
```

`0xbebafeca` or `FAT_CIGAM` is a 32-bit fat header in byte swapped (big-endian) format. This means that the `-e` is not necessary. Why is 68 bytes used for a length? Let's do the math…

- There's a `struct fat_header` right at the beginning containing two, 4-byte members called `magic` and `nfat_arch`. The `nfat_arch` has a value of `0x03000000`, but since you know it's byte-swapped, the actual value is `0x00000003`. This brings the total count to 8 bytes so far from this header.
- Immediately following the `fat_header` are three `struct fat_arch`s (because `nfat_arch` is 3). The `fat_arch` is the 32-bit equivalent which contains 5, 4-byte members. That means there's an additional 60 bytes (20-bytes × 3) of interest, which brings the total byte count to 68.

Augment the above Terminal command by replacing the `-e` argument with the byte size argument (`-g`), saying to display output in 4-byte groupings.

```
xxd -l 68 -g 4 /System/Library/Frameworks/CoreFoundation.framework/
CoreFoundation
```

Now the raw fat header data should be more viewable.

```
00000000: cafebabe 00000003 01000007 00000003  ................
00000010: 00001000 00747630 0000000c 00000007  .....tv0........
00000020: 00000003 00749000 0067cae0 0000000c  .....t...g......
00000030: 01000007 00000008 00dc6000 0074e6d0  .........`..t..
00000040: 0000000c                             ....
```

> **Note**: If the fat header was the 64-bit variant, the `-g 4` option wouldn't have worked since there are a couple of 8-byte variables mixed with 4-byte ones in `struct fat_arch_64`.

The first two values — `0xcafebabe` and `0x00000003` — are the `struct fat_header`, while the remaining bytes will belong to one of the three `struct fat_arch`s. Examining the first `struct fat_arch`, we can see it's for x86_64 due to the cputype `0x01000007` and cpusubtype `0x00000003` that you saw previously. The offset to the start of the x86_64 slice is `0x00001000` (4096) and whose size is `0x00747630`.

To prove that the x86_64 slice is at offset 4096 from the start of the file, dump the x86_64 header using `xxd`'s **-s** option.

```
xxd -l 32 -e -s 4096 /System/Library/Frameworks/CoreFoundation.framework/
CoreFoundation
```

As you can guess, the `-s` option specifies an offset to start at. You'll see the x86_64 slice's Mach-O header.

```
00001000: feedfacf 01000007 00000003 00000006  ................
00001010: 00000015 00001120 02100085 00000000  .... ............
```

Now that the headers are discussed, time to jump into the load commands.

The load commands

Immediately following the Mach-O header are the load commands providing instructions on how an executable should be loaded into memory, as well as other miscellaneous details. This is where it gets interesting. Each load command consists of a series of `structs`, each varying in struct size and arguments.

Fortunately, for each Load Command `struct`, the first two variables are always consistent, the **cmd** and the **cmdsize**. The `cmd` will indicate the type of load command and the `cmdsize` will give you the size of the `struct`. This lets you iterate over the load commands and then jump by the appropriate `cmdsize`.

The Mach-O authors anticipated this situation and provided a generic load command `struct` named **`struct load_command`**.

```
struct load_command {
  uint32_t cmd;   /* type of load command */
  uint32_t cmdsize; /* total size of command in bytes */
};
```

This lets you start with each load command as this generic `struct load_command`. Once you know the `cmd` value, you can cast the memory address into the appropriate `struct`.

So what are the values that `cmd` can have? Again, we put our faith in **mach-o/loader.h**.

```
#define LC_SEGMENT_64 0x19  /*64-bit segment of this file to be mapped*/
#define LC_ROUTINES_64  0x1a  /* 64-bit image routines */
#define LC_UUID   0x1b  /* the uuid */
```

If you see a constant that begins with `LC`, then you know that's a load command. There are 64-bit and 32-bit equivalents to load commands, so make sure you use the appropriate one. 64-bit load commands will end in a "`_64`" in the name. That being said, 64-bit systems can still use 32-bit load commands. For example, the `LC_UUID` load command doesn't contain the `_64` in the name but is included in all executables.

`LC_UUID` is one of the simpler load commands, so it's a great example to start out with. The `LC_UUID` provides the **Universal Unique Identifier** to identify a specific version of an executable. This load command doesn't provide any specific segment information, as it's all contained in the `LC_UUID` struct.

In fact, the load command struct for the `LC_UUID` load command is the **`struct uuid_command`** found in **mach-o/loader.h**:

```
/*
 * The uuid load command contains a single 128-bit unique random number
that
 * identifies an object produced by the static link editor.
 */
struct uuid_command {
    uint32_t  cmd;    /* LC_UUID */
    uint32_t  cmdsize;  /* sizeof(struct uuid_command) */
    uint8_t uuid[16]; /* the 128-bit uuid */
};
```

Going back to `grep`, you can view `grep`'s UUID using the **otool** command with the `-l` (load command) option.

```
otool -l $(which grep) | grep LC_UUID -A2
```

This will dump the two lines following any hits that contain the phrase `LC_UUID`.

```
      cmd LC_UUID
  cmdsize 24
     uuid 3B067B3F-4F1F-39A3-A4B9-CFDD595F9289
```

`otool` has translated the `cmd` from `0x1b` to `LC_UUID`, displays the `cmdsize` to `sizeof(struct uuid_command)` (aka 24 bytes) and has displayed the UUID value in a pretty format. If you're using the same macOS version as me, then you'll have the same UUID!

Segments

The `LC_UUID` is a simple load command since it's self-contained and doesn't provide offsets into the executable's segments/sections. It's now time to turn your attention to **segments**.

A segment is a grouping of memory that has specific permissions. A segment can have 0 or more subcomponents named **sections**.

Before going into the load command structs that provide instructions for segments, let's talk about some segments that are typically found in a program.

- The **__PAGEZERO segment** is a section in memory that is essentially a "no man's land." This segment contains 0 sections. This memory region doesn't have read, write, or execute permissions and occupies the lower 32-bits in a 64-bit process. This is useful in case the developer screws up a pointer, or dereferences `NULL`; if this happens, the program will crash since there's no read permissions on that memory region (bits 0 to 2^32). Only the main executable (i.e. not a framework) will contain a `__PAGEZERO` load command.
- The **__TEXT segment** stores readable and executable memory. This is where the application's code lives. If you want to store something that shouldn't be changed around in memory (like executable code or hard-coded strings), then this is the segment to put content in. Typically the `__TEXT` segment will have multiple sections for storing various immutable data.
- The **__DATA segment** stores readable and writable memory. This is where the majority of Objective-C data goes (since the language is dynamic and can change at runtime) as well as mutable variables in memory. Typically the `__DATA` segment will have multiple sections for storing various mutable data.
- The **__LINKEDIT Segment** is a grab-bag of content that only has readable permissions. This segment stores the symbol table, the entitlements file (if not on the Simulator), the codesigning information and other essential information that enables a program to function. Even though this segment has lots of important data packed inside, it has no sections.

Let's hammer this information in by looking at a real-life example. Use LLDB and attach to any process. Yes, any process! I'll choose the Simulator's **SpringBoard** for the example.

After starting up Simulator, type the following:

```
lldb -n SpringBoard
```

Once attached, type the following:

```
(lldb) image dump sections SpringBoard
```

This will dump the sections (and segments) for the `SpringBoard` module.

```
Sections for '/Applications/Xcode-beta.app/Contents/Developer/Platforms/
iPhoneOS.platform/Developer/Library/CoreSimulator/Profiles/Runtimes/
iOS.simruntime/Contents/Resources/RuntimeRoot/System/Library/
CoreServices/SpringBoard.app/SpringBoard' (x86_64):
  SectID     Type             Load Address
Perm File Off.  File Size  Flags      Section Name
  ---------- ---------------- ---------- ----------------------------
---- ---------- ---------- ---------- ----------------------------
  0x00000100 container        [0x0000000000000000-0x0000000100000000)*
---  0x00000000 0x00000000 0x00000000 SpringBoard.__PAGEZERO
  0x00000200 container        [0x000000010c0da000-0x000000010c8b4000)  r-
x  0x00000000 0x007da000 0x00000000 SpringBoard.__TEXT
  0x00000001 code             [0x000000010c0de69c-0x000000010c70d663)  r-
x  0x0000469c 0x0062efc7 0x80000400 SpringBoard.__TEXT.__text
... etc ...
```

Again, this is the in memory breakdown of the different Segments and Sections. If you wanted to see the on disk Mach-O layout instructions, you can use the following command:

```
(lldb) image dump objfile SpringBoard
```

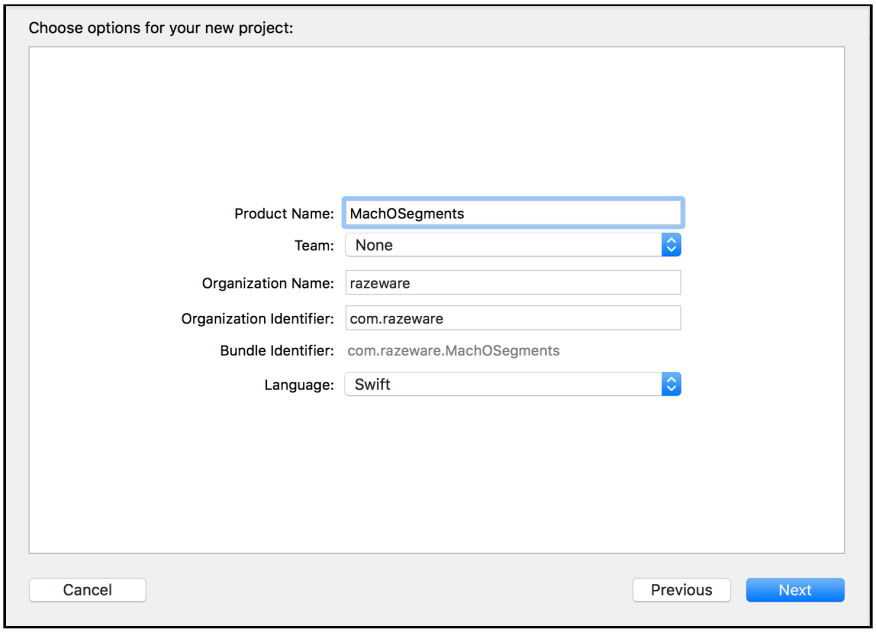
This will give you a fair bit of information since it spits out the Mach-O load commands as well as all the symbols found in the module. If you scroll to the top, you'll find all the load commands. It's important to note the difference between the on-disk location of the segment and sections, versus the actual memory location of the segment and sections, since they will have different values once loaded into memory.

> **Note**: And, as usual, I've written a LLDB command that displays the output in a much nicer format and has more advanced options for getting information out of specific sections in an executable. You can find it here: https://github.com/DerekSelander/LLDB/blob/master/lldb_commands/section.py

Programmatically finding segments and sections

For the demo part of this chapter, you'll build a macOS executable that iterates through the loaded modules and prints all the segments and sections found in each module.

Open Xcode, create a new project, select **macOS** then **Command Line Tool**, and name this program **MachOSegments**. Make sure the **Swift** language is selected.



Open **main.swift** and replace its contents with the following:

```
import Foundation
import MachO // 1

for i in 0..<_dyld_image_count() { // 2
  let imagePath =
    String(validatingUTF8: _dyld_get_image_name(i))! // 3
  let imageName = (imagePath as NSString).lastPathComponent
  let header = _dyld_get_image_header(i)! // 4
  print("\(i) \(imageName) \(header)")
}

CFRunLoopRun() // 5
```

Breaking this down:

1. Although `Foundation` will indirectly import the `MachO` module, you are explicitly importing the `MachO` module just to be safe and for code clarity. You'll be using several of the structs found in **mach-o/loader.h** in a second.

2. The **`_dyld_image_count`** function will return the total count of all the loaded modules in the process. You'll use this to iterate over all the modules in a for loop.
3. The **`_dyld_get_image_name`** function will return the full path of the image.
4. The **`_dyld_get_image_header`** will return the load address of the Mach-O header (`mach_header` or `mach_header_64`) for that current module.
5. The **`CFRunLoopRun`** will prevent the app from exiting. This is ideal, because I'll have you inspect the process with LLDB after the output is done.

Build and run the program. You'll see a list of modules and their load addresses spit out to the console. These load addresses are the location to where that particular Mach-O header resides in memory for that module. This is almost the exact same as doing a `image list -b -h` in LLDB! If you're curious and want one of these values to take a peek the Mach-O header, copy one of the values and use LLDB to dump the memory.

For example, in my output I see the following:

```
8 CoreFoundation 0x00007fff33cf6000
```

You can view the raw bytes of `CoreFoundations` Mach-O Header by pausing execution and typing the following in LLDB:

```
(lldb) x/8wx 0x00007fff33cf6000
```

And then you'll see something similar to the following:

```
0x7fff33cf6000: 0xfeedfacf 0x01000007 0x00000008 0x00000006
0x7fff33cf6010: 0x00000013 0x00001100 0xc2100085 0x00000000
```

Now you have the basic output in the `MachOSegments` program, add the following code to the end of the `for` loop:

```
var curLoadCommandIterator = Int(bitPattern: header) +
  MemoryLayout<mach_header_64>.size // 1
for _ in 0..<header.pointee.ncmds {
  let loadCommand =
    UnsafePointer<load_command>(
      bitPattern: curLoadCommandIterator)!.pointee // 2

  if loadCommand.cmd == LC_SEGMENT_64 {
    let segmentCommand =
      UnsafePointer<segment_command_64>(
        bitPattern: curLoadCommandIterator)!.pointee // 3

    print("\t\(segmentCommand.segname)")
  }

  curLoadCommandIterator =
```

```
    curLoadCommandIterator + Int(loadCommand.cmdsize) // 4
}
```

This is where the ugliness of Swift and C interopability really starts to rear its ugly head. Again with the numbers breakdown:

1. Load commands start immediately after the Mach-O header, so the header address is added to the size of the load address of the `mach_header_64` to determine where the load commands start. A good program would check if it's running a 32-bit mode by determining the `magic` value, but it's fun to walk on the wild side occassionally...
2. Using Swift's `UnsafePointer` to cast the load command to the "generic" `load_command` struct that you saw earlier. If this struct contains the correct `cmd` value, you'll cast this memory address to the appropriate `segment_command_64` struct.
3. Here you know that the `load_command` struct should actually be a `segment_command_64` struct, so we're using Swift's `UnsafePointer` object again.
4. At the end of each loop, we need to increment the `curLoadCommandIterator` variable by the size of the current `loadCommand`, which is determined by its `cmdsize` variable.

> **Note**: How did I know to cast the `segment_command_64` struct when I saw the value `LC_SEGMENT_64`? In the `mach-o/loader.h` header, search for all references to `LC_SEGMENT_64`. There's the declaration that defines `LC_SEGMENT_64` and then there's the `segment_command_64` which states its `cmd` is `LC_SEGMENT_64`. Finding all references to the load command will give you the appropriate C `struct`.

Build and run.

Upon execution, you'll get some rather ugly output like the one truncated one below.

```
0 MachOPOC 0x0000000100000000
  (95, 95, 80, 65, 71, 69, 90, 69, 82, 79, 0, 0, 0, 0, 0, 0)
  (95, 95, 84, 69, 88, 84, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0)
  (95, 95, 68, 65, 84, 65, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0)
  (95, 95, 76, 73, 78, 75, 69, 68, 73, 84, 0, 0, 0, 0, 0, 0)
```

This is because Swift is really terrible when working with C. `segmentCommand.segname` is declared as a Swift tuple of `Int8`s. This means you get to build a helper function to convert these values to an actual readable Swift `String`.

Jump to the top part **main.swift** and declare the following function.

```
func convertIntTupleToString(name : Any) -> String {
  var returnString = ""
  let mirror = Mirror(reflecting: name)
  for child in mirror.children {
    guard let val = child.value as? Int8,
      val != 0 else {
        break
    }
    returnString.append(Character(UnicodeScalar(UInt8(val))))
  }

  return returnString
}
```

Using the **Mirror** object, you can take a tuple of any size and iterate over it. It's much cleaner than hard coding to a parameter of type Tuple with 16 `Int8`'s.

Jump back down to the main body, and replace `print("\t\(segmentCommand.segname)")` with the following:

```
let segName = convertIntTupleToString(
  name: segmentCommand.segname)
print("\t\(segName)")
```

Build and run.

```
0 MachOPOC 0x0000000100000000
  __PAGEZERO
  __TEXT
  __DATA
  __LINKEDIT
1 libBacktraceRecording.dylib 0x0000000100ac7000
  __TEXT
  __DATA
  __LINKEDIT
2 libMainThreadChecker.dylib 0x0000000100ad7000
  __TEXT
  __DATA
  __LINKEDIT
  ...
```

Much better, right? Now each module will print out its containing Segments.

You're almost there! The final hurdle with print out the remaining Sections for each Segment.

Right below the new print command you just created, add the following code:

```
for j in 0..<segmentCommand.nsects { // 1
  let sectionOffset = curLoadCommandIterator +
    MemoryLayout<segment_command_64>.size // 2
```

```
    let offset = MemoryLayout<section_64>.size * Int(j) // 3
    let sectionCommand =
      UnsafePointer<section_64>(
        bitPattern: sectionOffset + offset)!.pointee

    let sectionName =
      convertIntTupleToString(name: sectionCommand.sectname) // 4
    print("\t\t\(sectionName)")
  }
```

The final round of numeric explanations:

1. In each `struct segment_command_64`, there's a member that specifies the number of `section_64` commands immediately following it. You'll use another `for` loop to iterate over all the sections found in each segment.
2. To start, you're grabbing the base address of the first `struct section_64` in memory.
3. For each iteration in the `for` loop, you'll start with the offset address then add the size of the `struct section_64` multipied by the iterator variable `j`. If you add the `sectionOffset + offset`, you'll get the correct `section_64` address to reference.
4. A `struct section_64` also has a **`sectname`** variable that is another tuple of `Int8`'s. You'll throw the same function you created earlier to grab a pretty Swift String out of it.

That's it for code. Build and run. Included is a tiny snippet of the output you'll get.

```
0 MachOPOC 0x0000000100000000
  __PAGEZERO
  __TEXT
    __text
    __stubs
    __stub_helper
    __cstring
    __objc_methname
    __const
    __swift4_types
    __swift4_typeref
    __swift4_reflstr
    __swift4_fieldmd
    __swift4_capture
    __swift4_assocty
    __swift4_proto
    __swift4_builtin
    __objc_classname
    __objc_methtype
    __swift4_protos
    __ustring
    __gcc_except_tab
    __unwind_info
    __eh_frame
  __DATA
```

```
      __nl_symbol_ptr
      __got
      __la_symbol_ptr
      __mod_init_func
      __const
      __cfstring
      __objc_classlist
      __objc_nlclslist
      __objc_catlist
      __objc_protolist
      __objc_imageinfo
      __objc_const
      __objc_selrefs
      __objc_protorefs
      __objc_classrefs
      __objc_superrefs
      __objc_ivar
      __objc_data
      __data
      __crash_info
      __thread_vars
      __thread_bss
      __bss
      __common
    __LINKEDIT
```

As you can see, only the main executable has the `__PAGEZERO` segment, which has 0 Sections. There's a slew of sections that contain `swift4` in them. There's a bunch of Objective-C related sections in the `__DATA` segment since Swift can't survive without Objective-C on Apple platforms.

In the next chapter, you'll look at some of these sections more closely and do some much more amusing things with the knowledge you got from this chapter.

Where to go from here?

If I haven't indirectly hinted it enough, go check out **mach-o/loader.h**. I've read that header many times myself, and each time I read it I still learn something new. There's a lot there, so don't get frustrated if this chapter knocked you back into your chair.

Play with all the variables with the structs you created. Check out the other load commands and match them with the appropriate structs. Add these commands to the demo project you created and see what information you can pull out of them.

Chapter 19: Mach-O Fun

Hopefully you're not too burnt out from dumping raw bytes and comparing them to `structs` in the previous chapter, because here comes the real fun!

To hammer in the different Mach-O section types, you'll build a series of examples across this chapter. You'll start with a "scanner" that looks for any insecure `http:` hardcoded strings loaded into the process. After that, you'll learn how to cheat those silly, gambling or freemium games where you never win the loot.

Commence funtime *meow*.

Mach-O Refresher

Just so you know what's expected of you, you'll start with a brief refresher from the previous chapter.

Segments are groupings on disk and in memory that have the same memory protections. Segments can have zero or more sections found inside a grouping.

Sections are sub-components found in a segment. They serve a specific purpose to the program. For example, there's a specific section for compiled code and a different section for hard-coded strings.

Sections and segments are dictated by the **load commands**, which are found at the beginning of the executable, immediately following the **Mach-O Header**

You saw a couple of the important segments in the previous chapter, notably the **__TEXT**, **__DATA**, and **__LINKEDIT** segments.

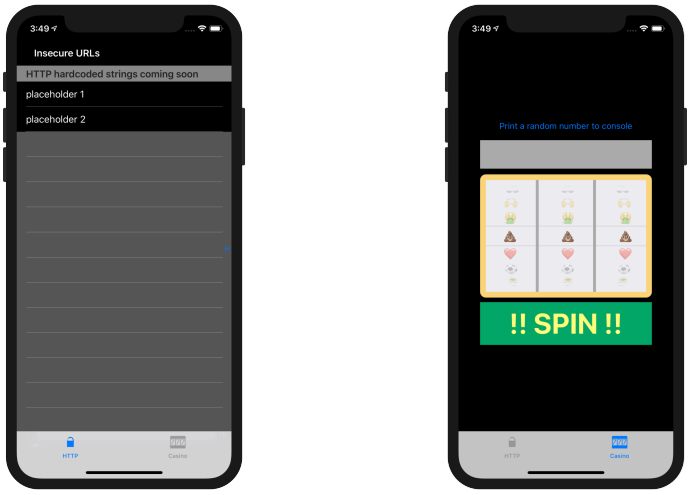
The Mach-O sections

Included in this chapter is an iOS project called **MachOFun**. Open it up and take a look around.

It's a simple `UITabBarController` application which breaks up the different examples you'll implement in this chapter.

One tab showcases a `UITableViewController` with some placeholder data. You'll first build a data source that finds all hardcoded insecure HTTP URLs in memory and then you'll display them in the `UITableView` as a "public shaming".

The other tab shows a rather ugly implementation of a slot machine for gambling purposes, for which you'll use your Mach-O knowledge to cheat the system and always win.



Build and run. At any point, suspend the program via LLDB and run the following command.

```
(lldb) image dump sections MachOFun
```

As you learned in the previous chapter, this will dump all the segments and corresponding sections found in the `MachOFun` module.

Search for the **`MachOFun.__TEXT.__text`**, section which stores executable code in the `MachOFun` application.

> **Note**: I am not the biggest fan of the `image dump sections [modulename]` LLDB command, since it produces an overload of output and is hard on the eyes. Also, if you forget to provide a module, LLDB will default to every module loaded into the process, which is a huge amount of output. But that command is the default and requires no extra setup. If you have trouble visually parsing the sections, use the LLDB console filter on the lower right of Xcode to make your life easier. Just remember to turn it off when you're done.

In the console output, you'll see the following:

```
0x00000001 code              [0x0000000109e5f290-0x0000000109e68aa0)  r-x
0x00001290 0x00009810 0x80000400 MachOFun.__TEXT.__text
```

- Breaking down this output, the **0x00000001** is LLDB's way to identify the section.
- LLDB has identified the content as **code**.
- The addresses in brackets is where this section is located in memory.
- The **0x00001290** is the offset on disk, while the **0x00009810** value is the size of the section on disk.
- Finally, the flags have the value **0x80000400** which are `S_ATTR_SOME_INSTRUCTIONS` and `S_ATTR_PURE_INSTRUCTIONS` OR'd together. Once again, I'll leave that to you to research in **mach-o/loader.h**.

> **Note**: The size of a section or segment on disk could be different when compared to the size when loaded into memory. This will be determined by the Mach-O load command. For example, the **`__PAGEZERO`** segment takes up 0 bytes on disk, but when loaded into memory, it takes up the first 2^32 bits in a 64-bit process. You can verify this on any executable (I use the `ls` as an example) by inspecting the Load Commands: **`otool -l $(which ls) | grep "Load command 0" -A11`**, The **filesize** variable is 0, while the **vmsize** variable is `0x0000000100000000`, or 2^32.

Now that you know how to parse the LLDB output, it's time to turn your attention back to `__TEXT.__text` section.

Using LLDB, take any method or function you can think of and find the section that it's located in. I'll use my go-to default, `-[UIViewController viewDidLoad]`, but you should pick something different.

```
(lldb) image lookup -n "-[UIViewController viewDidLoad]"
```

You'll see output similar to the following:

```
1 match found in /Applications/Xcode.app/Contents/Developer/Platforms/
iPhoneOS.platform/Developer/Library/CoreSimulator/Profiles/Runtimes/
iOS.simruntime/Contents/Resources/RuntimeRoot/System/Library/
PrivateFrameworks/UIKitCore.framework/UIKitCore:
        Address: UIKitCore[0x0000000000abb3f8] (UIKitCore.__TEXT.__text +
11243624)
        Summary: UIKitCore`-[UIViewController viewDidLoad]
```

The method `-[UIViewController viewDidLoad]` is located in **UIKitCore**. As mentioned previously, if you're running a version of iOS earlier than 12, then this method will be located in `UIKit`. The output above gives the full path to the `UIKitCore` module. The offset on disk is shown, via `UIKitCore[0x0000000000abb3f8]`; it's contained in `UIKitCore.__TEXT.__text` section at offset `11243624`. If you wanted the load address to be displayed in the output, then you could supply the `--verbose` option (or just `-v`) to LLDB.

Clear the screen and run the `image dump sections MachOFun` LLDB command again. This time, search for the **`MachOFun.__TEXT.__cstring`** Section.

In my output, I got the following:

```
0x00000006 data-cstr        [0x0000000109e6a320-0x0000000109e6b658)  r-x
0x0000c320 0x00001338 0x00000002 MachOFun.__TEXT.__cstring
```

This is where the UTF-8 hardcoded strings are stored for your print statements, key-value coding/observing, or anything else that's between `"`'s in your source code. Using LLDB, dump the memory in this section by referencing the size and the start load address.

```
(lldb) memory read -r -fC -c 0x00001338 0x0000000109e6a320
```

This prints the memory starting at address `0x0000000109e6a320`, format the output as printable characters (`-fC`), repeat the process `0x00001338` times, and force (`-r`) to print the entire count, since LLDB defaults to an upper limit of 1024 bytes. You'll see a load of familiar strings in here, such as `"Unexpectedly found nil while unwrapping an Optional value"`. Take a look through the strings and see what you can find.

Numerous other sections contain UTF-8 strings for different purposes. For example, the **`__TEXT.__objc_methname`** section contains Objective-C method names that are referenced directly by your application. The **`__TEXT.__swift4_reflstr`** section contains references to Swift's reflected items. A candidate for Swift runtime reflection would be references to `IBOutlet` or `IBInspectable` variables.

I highly recommend exploring these sections further on your own.

Printing unicode characters

The fun doesn't just stop at UTF-8 strings. In addition to the `__TEXT.__cstring` section, there's also the `__TEXT.__ustring` section which stores UTF-16 encoded strings.

Since these strings are UTF-16 encoded, you need to look at them with as appropriately sized characters.

Using the same `image dump sections` process as described above, find the load address of the `__TEXT.__ustring` located in memory. Once you have the start address, type the following in LLDB while remembering to use your own load address:

```
(lldb) exp -l objc -- (char16_t *)0x0000010a07ead8
```

You could get different output depending on what future version of `clang` decides to compile first, but you'll likely end up with an emoji character.

```
(char16_t *) $26 = 0x000000010a07ead8 u"💩"
```

This "pile of poop" emoji is one of the icons used in the slot machine's data source!

Finding HTTP strings

Now that you know hardcoded UTF-8 strings are stored in the `__TEXT.__cstring` module, you'll use that knowledge to search every module in the process to see if any string begins with the characters **`"http:"`**

Open up **InsecureNetworkRequestsTableViewController.swift** and add the following below `import UIKit`:

```
import MachO
```

Next, navigate to the **setupDataSource** function.

The logic is all setup to display any hits from the **dataSource** variable, but it only contains placeholer data for now. The `dataSource` variable is a typealiased array of `(module: String, strings: [String])`'s. That means for every element in the array, there will be a module name, plus an array of strings for that module that contain any insecure `"http:"` strings.

Remove the placeholder code in **setupDataSource** and replace it with the following:

```
for i in 0..<_dyld_image_count() {
    let imagePath = _dyld_get_image_name(i)!
    let name = String(validatingUTF8: imagePath)!
    let basenameString = (name as NSString).lastPathComponent

    var module : InsecureHTTPRequestsData = (basenameString, [])
    var rawDataSize: UInt = 0
    guard let rawData =
      getsectdatafromFramework(basenameString,
                               "__TEXT",
                               "__cstring",
                               &rawDataSize) else {
      continue
    }

    print("__TEXT.__cstring data: \(rawData), \(basenameString)")
}
```

The main point of interest in this code is the **getsectdatafromFramework** API. This function takes the name of the module, the name of the containing segment as well as the section and gives the pointer to the location of the section in memory! In addition, there's an `inout` variable called `rawDataSize` which gives the size of the section in memory.

Build and run. You'll see every module that contains a `__TEXT.__cstring` section as well as the appropriate load address of where in memory it can be found.

From the console output, you can see the following:

```
__TEXT.__cstring data: 0x00000001025c35a0, dyld_sim
__TEXT.__cstring data: 0x000000010256c2f0, MachOFun
...
```

Pause the application. Then take any address you find and use LLDB to query information about. I'll take the `__TEXT.__cstring` load address for `dyld_sim` in my process. As always, your output for load addresses will likely be different.

Grabbing the `__TEXT.__cstring` load address of `dyld_sim`, query info about it using LLDB:

```
(lldb) image lookup -a 0x000000010256c2f0
```

I get the following output:

```
Address: dyld_sim[0x0000000000365a0] (dyld_sim.__TEXT.__cstring + 0)
Summary: "__LINKEDIT"
```

The `dyld_sim[0x0000000000365a0]` shows the offset on disk to where the `__TEXT.__cstring` location is stored. Remember, that might not be the finalized offset on disk if the executable is a fat executable with multiple architecture slices. The `Summary` part of the output might seem a little misleading with `"__LINKEDIT"`, but remember, this is the location of where hardcoded UTF-8 strings are stored. Using LLDB, print out the first string at `dyld_sim.__TEXT.__cstring`, like so:

```
(lldb) x/s 0x000000010256c2f0
```

You'll get

```
0x1025c35a0: "__LINKEDIT"
```

The first hardcoded string compiled into the `dyld_sim` module is the string `"__LINKEDIT"`. This is the output of the compiled version of `dyld_sim` on my machine, and the first string could be different in other versions of `dyld_sim`.

Now that you've found the start address of the `__TEXT.__cstring` sections, it's time to parse that whole buffer of memory to search for any strings that begin with `"http:"`.

Remember, this buffer of memory is a bunch of UTF-8 C strings. That means you need to parse a string for as long as you can until you hit a NULL byte.

Open **InsecureNetworkRequestsTableViewController.swift** and in `setupDataSource()`, remove the `print` statement you made earlier and replace with the following:

```
var index = 0
while index < rawDataSize {
  let cur = rawData.advanced(by: index)
  let length = strlen(cur)
  index = index + length + 1

  guard let str = String(utf8String: cur),
    length > 0 else {
      continue
  }

  if str.hasPrefix("http:") {
    module.strings.append(str)
  }
}

if module.strings.count > 0 {
  dataSource.append(module)
}
```

This code will grab the `rawData` pointing to a `__TEXT.__cstring` section in memory. The `while` loop performs several checks, making sure a valid UTF-8 string of length greater than 0 exists. If so, then the beginning of the string is checked to see if it contains the characters `"http:"`. If so, then the string is added to the `strings` array. Finally, if a module has any strings that contain `"http:"`, then that is added to the `dataSource` variable.

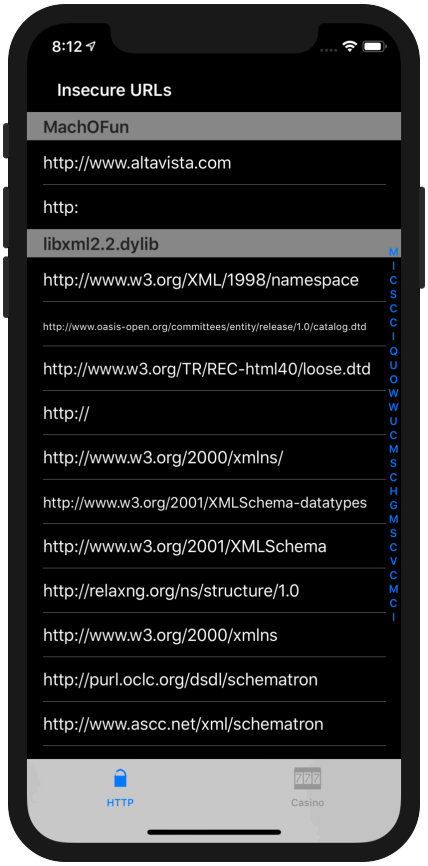
Finally, make sure you have a controlled test in the `MachOFun` module to make sure this is correctly working.

In `viewDidLoad`, add the following code:

```
let _ = "https://www.google.com"
let _ = "http://www.altavista.com"
```

If everything works as expected, the `https://www.google.com` string will not be displayed (since it begins with "https"), while the `http://www.altavista.com` string will (hopefully?) be displayed.

Build and run.



As you can see, there are a number of insecure hardcoded URLs not only in the `MachOFun` module, but modules like `libxml2.2dylib`, `GeoServices`, `CFNetwork`, etc.

What's up with CFNetwork's strings?

This oddity you can optionally explore as this is a smidge outside the context of the chapter, but `CFNetwork` has some amusing hardcoded strings embedded in the module.

Modify the `if` conditional code you just wrote:

```
if str.hasPrefix("http:") {
```

To the following:

```
if str.contains("RubberDucky") {
```

Where **RubberDucky** is replaced by your favorite English curse word. I promise you won't be disappointed as `CFNetwork`'s (potentially Unit Testing?) engineers have an interesting sense of humor of websites to test their framework on.

Sections in the __DATA segment

Now that you've got your public insecure URL shaming out of the way, it's time to shift the attention to the writeable `__DATA` segment and explore some interesting sections.

Suspend the `MachOFun` app and use LLDB to query the data sections. Execute the good ol' following LLDB command:

```
(lldb) image dump sections MachOFun
```

Search for the **__DATA.__objc_classlist** section in the output. In my process, I got the following...

```
0x00000016 data-ptrs        [0x000000010dcae8e0-0x000000010dcae900)  rw-
0x0000f8e0 0x00000020 0x10000000 MachOFun.__DATA.__objc_classlist
```

This section stores `Class` pointers to Objective-C or Swift classes. This section is an array of `Class` pointers that point to the actual `Classes` stored into **`__DATA.__objc_data`**. Think of the `__DATA.__objc_data` section as a buffer of Objective-C data packed together, just as how the hardcoded UTF-8 strings are stored in the `__TEXT.__cstring` section.

Jumping back to the `__DATA.__objc_classlist` section, you can quickly determine that there are four classes implemented by the `MachOFun` module. How can you determine this? The segment size is `0x00000020` divided by the size of a pointer in a 64-bit process (8 bytes), which leaves you with four Objective-C/Swift classes.

Use LLDB to dump the raw pointers from the `__DATA.__objc_classlist` section to prove this is correct.

```
(lldb) x/4gx 0x0000010dcae8e0
0x10dcae8e0: 0x000000010dcb0580 0x000000010dcb0690
0x10dcae8f0: 0x000000010dcb0758 0x000000010dcb0800
```

Then for each pointer:

```
(lldb) exp -l objc -O -- 0x000000010dcb0580
MachOFun.CasinoContainerView

(lldb) exp -l objc -O -- 0x000000010dcb0690
MachOFun.CasinoViewController

(lldb) exp -l objc -O -- 0x000000010dcb0758
MachOFun.InsecureNetworkRequestsTableViewController

(lldb) exp -l objc -O -- 0x000000010dcb0800
MachOFun.AppDelegate
```

Inside `MachOFun`, there are four Swift classes, due to the fact the module and period precedes the class name.

Tools like **`class-dump`** (available here http://stevenygard.com/projects/class-dump/) use this information along with numerous other Mach-O sections to display Swift/Objective-C classes.

The __bss, __common, and __const sections

Sometimes a module needs to keep references to data that lives past a function call. As you've learned earlier, if you were to declare a constant such as `let v = UIView()` inside of a function, the pointer `v` is stored on the stack which points to allocated memory on the heap. But as soon as the instruction pointer leaves the function, the reference to the `v` variable is long gone. That's why there are several sections in the `__DATA` segment designed to store variables across the lifetime of a process.

When you declare a **global variable**, which is a variable outside the scope of any method or function, it will typically be placed into the **`__DATA.__common`** section. This section expects to share information across the module and even across other modules.

What if a developer wanted to have a variable survive across function calls, but not have it accessible to any other modules, or even from other source files within the same module? This is typically achieved by storing variables in the **__DATA.__bss** section. The C/Objective-C family will do this via a **static** declaration to a variable. In Swift, this can be achieved with a **private** declaration on a Swift variable.

Finally, there are global variables that you want declared as unchanging for the life of the program. You can mark these as **const** in C/Objective-C to store variables in the **__DATA.__const** section. From a developers standpoint, Swift (mostly) doesn't need you to touch the __DATA.__const section due to the let keyword and checking for changes to a variable at compile time.

Cheating freemium games

The __DATA segment not only stores references to data in the module, but it also provides references to external variables, classes, methods, and functions that are not defined within the module.

Think about why this is the case for a second. If, in theory, a module can be loaded at any address, a reference point must be used to indicate where to start looking when calling out to that code. Since this location is not known until runtime, this starting reference point *must* be writable from the calling module.

This applies to external C functions, Swift/Objective-C classes, global variables, etc.

The **__DATA.__la_symbol_ptr** is a rather interesting section as it stores references to external functions that are lazily resolved at runtime when called. For this complex dance to work, the __DATA.__la_symbol_ptr section stores a series of function pointers that point into offsets into the **__TEXT.__stub_helper** section in the calling module. This sets of a flurry of activity as dyld resolves the location of this external function. I'll stay out of the gory details of this, but just know that external functions by default are referenced through the __DATA.__la_symbol_ptr section and are "resolved" if the function pointer doesn't point to an address in the __TEXT.__stub_helper section.

Resume execution of the MachOFun program and navigate the app to the Casino tab. Once at the slot machine, give it a couple of spins.

For you intermediate to advanced readers out there, see if you can recall the API or APIs to generate a random number. Remember your guess and see if it's true below. Suspend the program and type the following in LLDB:

```
(lldb) exp -l objc -O -- [[NSBundle mainBundle] executablePath]
```

This will give you the full path to the running application. Copy the full path to the clipboard and type the following in LLDB:

```
(lldb) platform shell objdump -lazy-bind ${APP_PATH} | grep $
{RANDON_NUMBER_FUNC}
```

You're running the Terminal command **`objdump`**, searching for all lazy bound functions inside the `MachOFun` executable.

Be sure to replace the `${APP_PATH}` with the actual path and your best guess for the random number generator function with `${RANDON_NUMBER_FUNC}`.

Did you guess the function correctly? I had to type the following:

```
platform shell objdump -lazy-bind /Users/derekselander/Library/Developer/
CoreSimulator/Devices/3C657090-3EF0-4A35-B202-07A8D2C85556/data/
Containers/Bundle/Application/2EB74DAB-09F7-4825-BEDE-5AB40335AE13/
MachOFun.app/MachOFun | grep arc
```

This gave me the following output:

```
__DATA   __la_symbol_ptr    0x10000F3F8 libSystem
_arc4random_uniform
```

This means that **`arc4random_uniform`** is being called somewhere in the `MachOFun` code. This function will generate a random number with a range given by the first parameter.

This **0x10000F3F8** value is the calculated offset in memory without the ASLR slide. This `0x10000F3F8` value includes the `__PAGEZERO` offset (given by the 0x100000000) with the actual *real* offset on disk with the value `0xF3F8`.

How can you translate this `0x10000F3F8` value into memory? You can use the **`_dyld_get_image_vmaddr_slide`** API to get the address slide!

In LLDB, type the following:

```
(lldb) po (char *)_dyld_get_image_name(1)
```

This is to make sure you are referencing the correct index into the modules. Typically index **1** will point to the main executable.

Make sure the output references the `MachOFun` executable.

```
"/Users/derekselander/Library/Developer/CoreSimulator/Devices/
3C657090-3EF0-4A35-B202-07A8D2C85556/data/Containers/Bundle/Application/
2EB74DAB-09F7-4825-BEDE-5AB40335AE13/MachOFun.app/MachOFun"
```

After that, use LLDB on index 1 with the `_dyld_get_image_vmaddr_slide` API and add it to the value you retrieved from **objdump** command:

```
(lldb) p/x (intptr_t)_dyld_get_image_vmaddr_slide(1) + 0x10000F3F8
```

For me, I got the value **0x000000010f91c3f8**. This value is the resolved load address to the location to the external stub reference of `arc4random_uniform` *in memory*. Dereference the value of this address and examine it with LLDB:

```
(lldb) x/gx 0x000000010f91c3f8
```

This will produce something similar to the following:

```
0x10f91c3f8: 0x000000011140f027
```

Query this new address and see what it resolved to:

```
(lldb) image lookup -a 0x000000011140f027
```

And lo and behold you'll get the in-memory address to `arc4random_uniform`:

```
Address: libsystem_c.dylib[0x0000000000025027]
(libsystem_c.dylib.__TEXT.__text + 144919)
Summary: libsystem_c.dylib`arc4random_uniform
```

This means that the `arc4random_uniform` function has already been resolved, since the function pointer in `__DATA.__la_symbol_ptr` is pointing to `arc4random_uniform` instead of an offset in the `__TEXT.__stub_helper` section.

Hell, you're not even going to set a breakpoint on `arc4random_uniform` since you're so confident that this slot machine is calling `arc4random_uniform` to generate random numbers. You'll change around the pointer in memory just to see what can happen!

In LLDB, create a global function that always returns the value 5.

```
(lldb) exp -l objc -p -- int lolzfunc()  { return 5; }
```

The out-of-the-ordinary `-p` option says to execute this code outside of any stack frame. This is necessary since you can't declare functions inside other C code. This means there's a global function named **lolzfunc** floating around somewhere in memory.

Grab the address of the `lolzfunc` via LLDB:

```
(lldb) p/x lolzfunc
(int (*)()) $14 = 0x000000012d4c7430 (0x000000012d4c7430)
```

The plan of attack should be clear now. You will change around the external stub's pointer of `arc4random_uniform` to the address of the newly created function, `lolzfunc`.

In LLDB, type the following:

```
(lldb) memory write -s8 0x000000010f91c3f8 0x000000012d4c7430
```

The first pointer is the original address of `arc4random_uniform` that you found earlier. The second pointer is the new `lolzfunc` address.

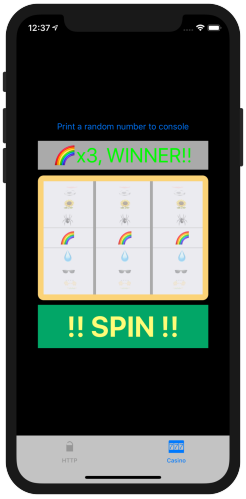
This tells LLDB to write 8 bytes (`-s8`) at location `0x000000010f91c3f8`, with value `0x000000012d4c7430`.

You just followed a very complex set of instructions. Here's an image to help you out to make sure you did everything as expected.

```
(lldb) # Get the fullpath
(lldb) ex -l objc -O --  [[NSBundle mainBundle] executablePath]
/Users/derekselander/Library/Developer/CoreSimulator/Devices/3C657090-3EF0-4A35-B202-07A8D2C85556/data/
  Containers/Bundle/Application/83941ACC-1E48-47A4-852B-FC7B1D27FE33/MachOFun.app/MachOFun

(lldb) # Is it the arc4.* APIs?
(lldb) platform shell objdump -lazy-bind /Users/derekselander/Library/Developer/CoreSimulator/Devices/
  3C657090-3EF0-4A35-B202-07A8D2C85556/data/Containers/Bundle/Application/83941ACC-1E48-47A4-852B-
  FC7B1D27FE33/MachOFun.app/MachOFun | grep arc
__DATA   __la_symbol_ptr    0x10000F3F8 libSystem        _arc4random_uniform
(lldb) # Get the resolved address in memory
(lldb) p/x (intptr_t)_dyld_get_image_vmaddr_slide(1) + 0x10000F3F8
(long) $1 = 0x000000010f91c3f8
(lldb) # Create the cheating function
(lldb) exp -l objc -p -- int lolzfunc()  { return 5; }
(lldb) # Get the address of cheating function
(lldb) p/x lolzfunc
(int (*)()) $2 = 0x000000012d4c7430 (0x000000012d4c7430)
(lldb) # write the new value into the arc4random_uniform stub
(lldb) memory write -s8 0x000000010f91c3f8 0x000000012d4c7430
(lldb)
```

Resume the application, then give the game another spin and see what happens.



You're winning... every time... what are the odds of that? Crazy, eh?

Objective-C swizzling vs function interposing

Unlike Objective-C method swizzling, lazy pointer loading occurs on a **per-module basis**. That means that the trick you just performed will only work when the `MachOFun` module calls out to `arc4random_uniform`. It wouldn't work if, say, `CFNetwork` called out to `arc4random_uniform`.

Going back to the MachoFun app, do you see that `"Print a random number to console"` button?



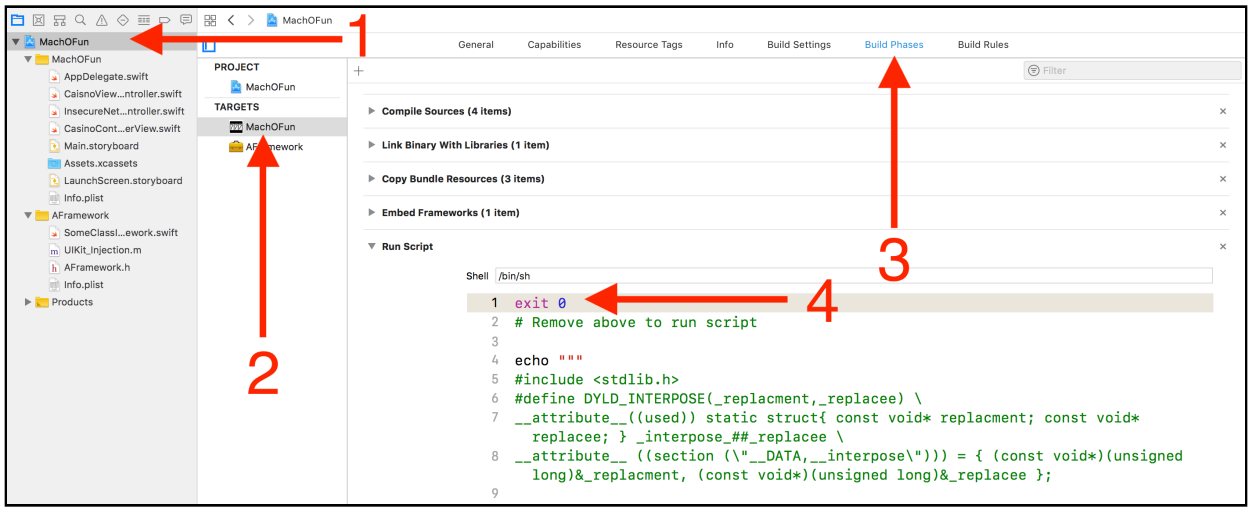
That code resolves to an `IBAction` method which calls `SomeClassInAFramework.printARandomNumber()`. That code is implemented in a different framework creatively called **AFramework**. Inside the static **printARandomNumber()** function, `arc4random_uniform` is being called.

Press the button a couple of times and observe how `arc4random_uniform` works normally. This means that if you wanted to swap all `arc4random_uniform` stubs, you'd have to iterate through each module, find the `arc4random_uniform` location stub in memory and replace it with the address of the new function.

Or do you? Fortunately, `dyld` will look for a special `__DATA` section in your executable called **`__DATA.__interpose`**. If `dyld` sees this section, it will replace all stubs with the new function in all modules loaded thereafter!

Quietly hidden away is a crippled shell script in the `MachOFun` target in Xcode. Click on the `MachOFun` project, select the `MachOFun`, select **Build Phases** and then expand the **Run Script**.

From there, remove the `exit 0` line right at the top of the build script.

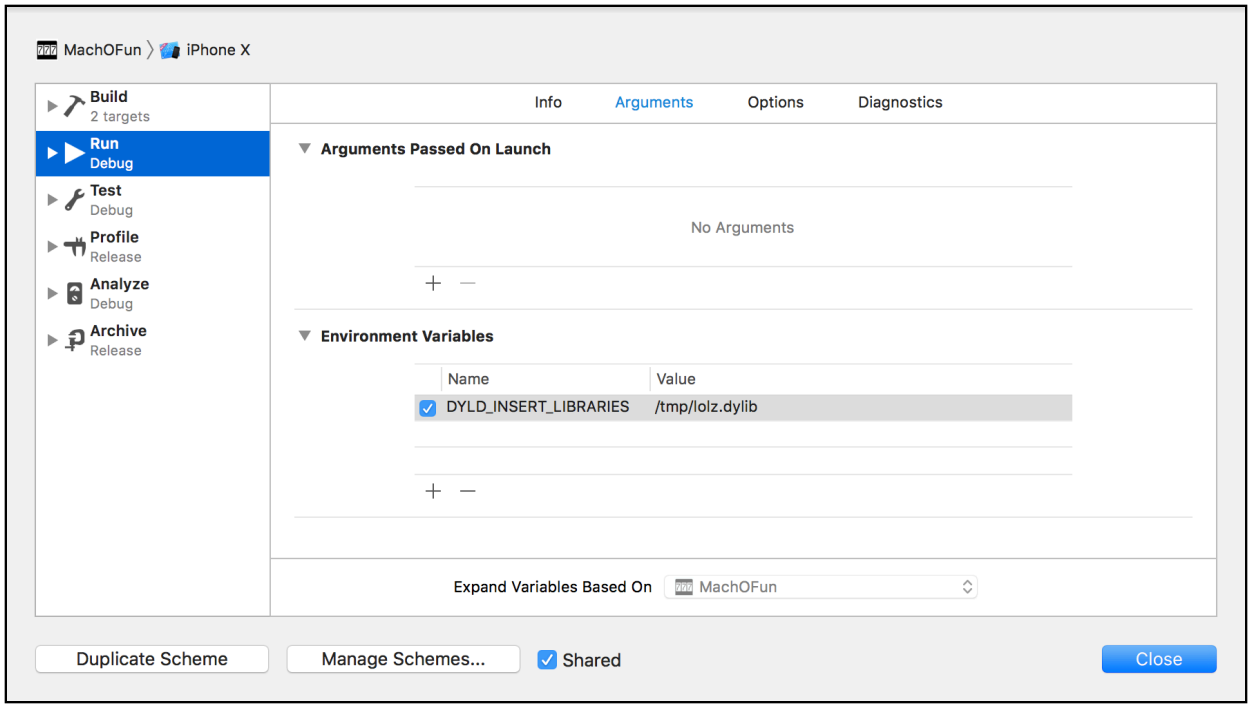


This script will create a dynamic library at **/tmp/lolz.dylib** and sign the library using an ad-hoc code signature, as the Simulator needs a valid signature even if the code signatures don't match.

This dynamic library is as simple as can be. It declares a function called **`dereksfunc`** which returns the integer 7. The address of this function is interposed with `arc4random_uniform` so all future modules will have these stubs replaced.

Using the powerful **DYLD_INSERT_LIBRARIES** environment variable will let you load this `lolz.dylib` module first, before any other modules are loaded, resulting in every single `arc4random_uniform` referenced by an external module to be replaced with `dereksfunc`

Navigate to the **Edit Scheme** window with ⌘ + **Shift** + <. Then select **Run** from the bar on the left, and then the **Arguments** tab on the right. Finally enable the already filled-out path to the `DYLD_INSERT_LIBRARIES` environment variable. Then click **Close**.



Build and run. Once active, navigate to the Casino tab in the Simulator and give the slot machine a couple of spins. You'll now consistently get the 🕶 slot since all `arc4random_uniform` calls return 7. Even the `"Print a random number to console"` button will consistently return 7.

Pause the app using LLDB and type the following:

```
(lldb) image dump sections lolz.dylib
```

Search for the following section:

```
0x00000003 regular          [0x000000010b8bc000-0x000000010b8bc010)  rw-
0x00001000 0x00000010 0x00000000 lolz.dylib.__DATA.__interpose
```

You'll see that the size is big enough for two function pointers (`0x00000010`).

If you dereference the start address with a `x/2gx 0x000000010b8bc000`, what values do you think are there?

Pretty cool!

Where to go from here?

Oh my! There is so much more for you to learn about Mach-O, but the road ends here for now.

- Check out Jonathan Levin's work on describing Mach-O in http://newosxbook.com/articles/DYLD.html.
- I also haven't even started on the complexity and power of the `__LINKEDIT` segment. A surprisingly good reference is Facebook's **Fishhook**, a runtime library for modifying external stubs, found here: https://github.com/facebook/fishhook. You will have a brief glimpse into the `__LINKEDIT`'s symbol table in the next chapter, but there will be a lot of information that can be learned elsewhere.
- I'd also recommend using `jtool` to learn how programs are built. Find an interesting Terminal command (like `ps`), and run the following command, will show all the functions that `ps` references:

  ```
  jtool -d __DATA.__la_symbol_ptr $(which ps)
  ```

  This is probably one of the best ways to learn about some of the lower level, more obscure functions.
- Finally, the last thing you can do is use your newfound knowledge of function interposition and stub binding to cripple all XCTest logic to make it work in your unit tests. That is, go impress your superior by "Volkswagon"'ing your unit tests and get 0 failed test cases!

Chapter 20: Code Signing

Ah, code signing: an iOS developer's nemesis. Code signing is hardly at the top of every iOS developer's agenda, but a strong knowledge of how code signing works can be extremely useful for solving problems, as well as establishing yourself as a lichpin in your development team. There's nothing more "ask-for-a-raise-worthy" than a developer who can re-sign an outdated Swift 2.2 iOS app, instead of having to fix the potentially thousands of Swift compiler errors when time is against you.

This chapter will give you a basic overview of how code signing works by having you pick apart the open-source iOS Wordpress v10.9 application found here:

- https://github.com/wordpress-mobile/WordPress-iOS/releases/tag/10.9



You'll explore the stages of the app's code signature before being sent to the iTunes Connect Store. In addition, you'll re-sign the Wordpress app so it can run on your very own iOS device!

Setting up

In order to complete everything in this chapter, you'll need a number of items. First, you'll need a proper **iOS Apple Developer account** to generate Provisioning Profiles. You'll also need a physical iOS device to install the `Wordpress` iOS application.

You'll also need to grab and install my **mobdevim** command, available here:

- https://github.com/DerekSelander/mobdevim

This small command line tool does a number of things, including installing applications onto the device, querying device information and grabbing console logs when connected to an iOS device. Using this tool makes it significantly easier to install an iOS app on your device and hunt for errors if the app has been incorrectly signed. You can install iOS apps in Xcode through the **Devices and Simulators** window, but the Xcode authors really make it a painful to do this.

Follow the instructions on the `mobdevsim` repo to install the tool. Once you have `mobdevim` setup, plug in your iOS device via cable and give it a test run.

```
mobdevim -f
```

This will query the device info. Provided it worked, you'll get something similar to:

```
Connected to: "LOLz" (c3a8533d6dc4fa74d6748a2cd935f00e1e949af1)

name    LOLz
UDID    c3a8533d6dc4fa74d6748a2cd935f00e1e949af1

State   Activated
Type    iPhone
Version 12.0.0
Number  (555) 867-5309
Region  LL/A
Battery 65%
... truncated ...
```

You can get a full list of the available commands by executing `mobdevim -h`.

After installing this command, and making sure you have a valid way to sign your iOS applications with a developer account, you'll be all set up for the rest of this chapter!

Terminology

To really appreciate how code signing works, you need to understand three key components: **public/private keys**, **entitlements**, and **provisioning profiles**. You'll start with a breadth-first look, then dive into a depth-first look at each.

The **public/private key** is used to sign your application. This is your digital signature which Apple knows about and how Apple verifies you as, well, you. The private key is used to crytographically sign the app as well as its capabilities, or **entitlements**. The entitlements are really just an XML string embedded in your app which says what the app can, and can't, do.

The grouping of the entitlements, the list of approved devices, and the public key to verify the code signature are all bundled together in a **provisioning profile** in your application. All the information is enforced through the signature created by your private key and can be verified by your public key.

That is a lot to take in, and it gets more confusing from here. So let's investigate each component of the profile separately.

Public/private keys

This is probably the hardest thing to understand when learning about the code signing process, since public/private keys introduce cryptography, which quickly becomes a rabbit hole of knowledge, formats, formatting, and gotchas.

Simply put, there's two different types of cryptography: **symmetric cryptography**, and **asymmetric cryptography**.

Symmetric cryptography is a type of cryptography that contains only one key. If person A tries to send a secret message to person B, they both must know that shared secret in order to encrypt and decrypt the message.

In asymmetric cryptography, there are two keys: a public key (which can be known by everyone) and a private key (which is kept secret to you). Both person A and B have their own unique private key and their own unique public key. That way, they can share information without either person knowing the other person's private key. The implementation of this is beyond the scope of this chapter, but you should learn more about this concept on your own time if this is new to you.

If you can remember when you set up your Apple developer account, you went through the process of **Requesting a Certificate From A Certificate Authority**. You created a public/private key, sent up the public key to Apple servers (by the `.csr` file). The end result of this process created a signature that is signed by Apple and is how Apple uniquely recognizes you. This means that Apple — and by extension, you — use asymmetric cryptography for distributing applications.

You can view the names, or **identities**, of your public/private key pairs used for signing your applications with the following Terminal command:

```
security find-identity -p codesigning -v
```

This command queries the macOS system keychain, looking for valid identities that contain a private key (`-v`) and whose type can codesign (`-p codesigning`).

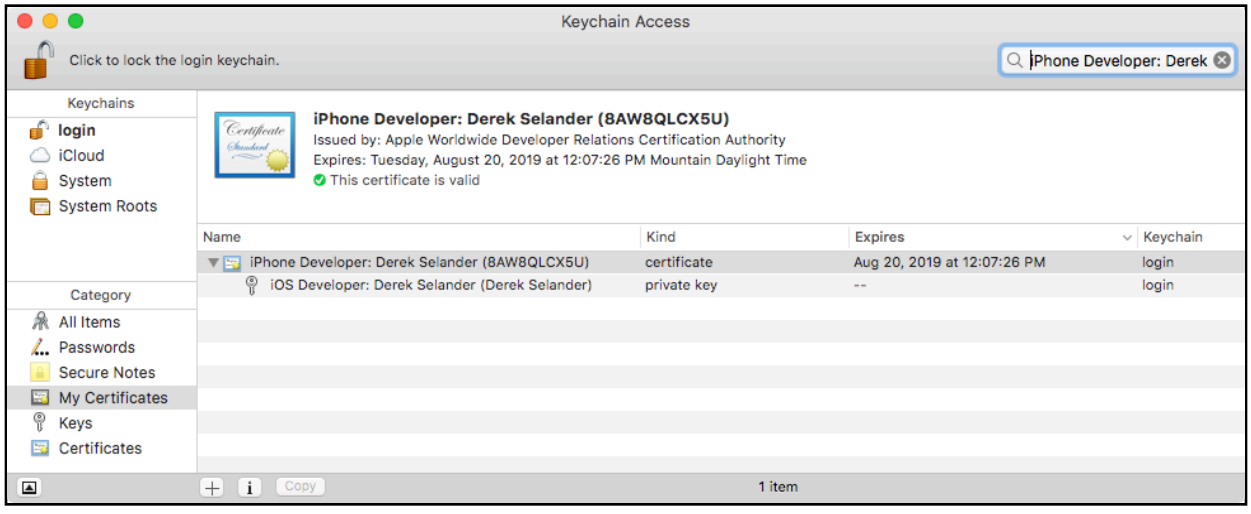
This ouput will display identities that are valid, which can produce a code signed application. If you look for identities that contain the phrase "iPhone Developer], it's likely that this identity can be used to sign an iOS application on your device.

For example, I got the following output for identities that contained the term "iPhone Developer":

```
1) 2DFE888B7BD07710444C2E4A7B9847BA8B55C220 "iPhone Developer: Derek
Selander (8AW8QLCX5U)"
```

If you got something similar, your computer is properly set up to sign a valid iOS application on your macOS machine.

You can view this identity in the GUI-equivalent program **Keychain Access**. Open Keychain Access, navigate to **My Certificates**, then search for your equivalent string by omitting the quotes.



Notice that you havea public key, or a **certificate**, as well as the **private key** found below. Certificates can be recreated, but private keys are worth more than gold. Never, ever, delete a private key! If you do, you forefit your proof that you're you, and you'll need to recreate a new identity through Apple.

Let me revisit a point you might have missed in the above paragraph. A certificate, in this sense, is only the **public key**. So if you were to use Keychain Access to export your identity, and you wanted to format it in a **.cer** (certificate) format, you'd only be exporting the public key. If you want to export the private key as well, you must use the PKCS12 format (**.p12**) to properly export the full identity, private key and all.

This is important to know if you wanted to export the identity so another developer could, say, generate a build with a matching distribution (i.e. App Store) identity. But be careful: whoever has the private key can assume the full identity for that company, at least from Apple's perspective!

Jumping back to the Terminal equivalent, you can export the public certificates using the following command:

```
security find-certificate -c "iPhone Developer: Derek Selander
(8AW8QLCX5U)" -p
```

This will output the public, x509 certificate of **iPhone Developer: Derek Selander (8AW8QLCX5U)** to `stdout` and format it in **PEM** format. There's two ways to display a certificate: DER and PEM. PEM can be read by the Terminal (since it's in base64 encoding) while DER, in highly professional coding terms, will produce gobbledygook and make the Terminal beep a lot.

Repeat the above command and write the output to `/tmp/public_cert.cer`. Be sure to replace the identity with your own identity:

```
security find-certificate  -c "iPhone Developer: Derek Selander
(8AW8QLCX5U)"  -p > /tmp/public_cert.cer
```

Use the Terminal command to `cat` this newly created file:

```
cat /tmp/public_cert.cer
```

You'll see something similar to:

```
-----BEGIN CERTIFICATE-----
MIIFnDCCBISgAwIBAgIIFMKm2AG4HekwDQYJKoZIhvcNAQELBQAwgZYxCzAJBgNV
BAYTAlVTMRMwEQYDVQQKDApBcHBsZSBJbmMuMSwwKgYDVQQLDCNBcHBsZSBXb3Js
ZHdpZGUgRGV2ZWxvcGVyIFJlbGF0aW9uczFEMEIGA1UEAww7QXBwbGUgV29ybGR3
...
```

This is how you can tell this certificate is in PEM. Terminal isn't cranky, and the header `-----BEGIN CERTIFICATE-----` is included. This would not be the case if the certificate was in DER format.

From here, you can use the `openssl` Terminal command to query the public, x509 certificate:

```
openssl x509 -in /tmp/public_cert.cer -inform PEM -text -noout
```

Yes, that's a lot of params!

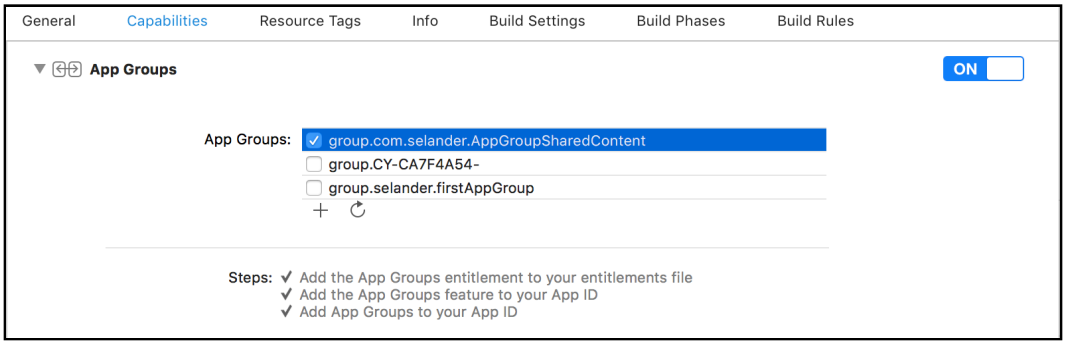
- The `x509` option says that the `openssl` command should be able to work with a `x509` certificate.
- You provide the `-in` to the path of **public_cert.cer** with the decoding format of PEM (`-inform PEM`).
- You specify you don't want to output a certificate with the `-noout` param.
- But instead, you do want the certificate in a (somewhat) readable "text" format with the `-text` option.

The information about this public certificate will be displayed in the Terminal.

Remember this `openssl` command, as you'll revisit the concept of x509 certificates when you read about the provisioning profiles which embed these public certificates inside of them.

Entitlements

Embedded in (almost) every compiled application is a set of **entitlements**: again, this is an XML string embedded in the application saying what an app can and can't do. Other programs will check for permissions (or lack thereof) in the entitlements and grant or deny a request accordingly. Think of the **capabilities** section found in Xcode.



Many of these permission checks are carried out by other daemons which check your programs entitlements. For example, **App Groups**, **iCloud Services**, **Push Notifications**, **Associated Domains** all will modify the entitlments to your app. These capabilities shown in Xcode are but a small piece of the entitlements on Apple platforms as the majority of them are private to Apple and enforced through code signing.

You can see a complete list of entitlements found in the wild thanks to Jonathan Levin's entitlement database, here:

- http://newosxbook.com/ent.jl

Probably the most important entitlement, at least in this book, is the **get-task-allow** entitlement, found on all your software compiled with a developer certificate. This allows the program in question to be attached to a debugger.

On macOS, you can get around the lack of this entitlement by disabling SIP for any applications that don't have the `true` value for this key. On iOS, you'll be S.O.L. trying to debug an application that doesn't have this entitlement, unless code verification has been disabled through jailbreaking.

You can view the entitlements of an application through the **`codesign`** Terminal command.

Find the entitlements of the macOS **Finder** application:

```
codesign -d --entitlements :- /System/Library/CoreServices/Finder.app/
Contents/MacOS/Finder
```

The `-d` option says to display the option immediately following in the command, which is the `--entitlements`. You also have that weird looking `:-`, which does two things:

- The `-` says to print to `stdout`
- The `:` says to omit the blob header and length.

Just as in Mach-O, the code signature information is stored with a magic header, immediately followed by a length. The "`:`" says to strip this header information out of the output and *only* display the actual XML string of entitlements.

Provisioning profiles

Finally, the provisioning profiles are up for discussion. A provisioning profile includes the public x509 certificate, the list of approved devices, as well as the entitlements all embedded into one file.

The default location for provisioning profiles can be found here:

```
~/Library/MobileDevice/Provisioning Profiles/
```

Unfortunately, provisioning profiles are named by their UUID instead of by the name you (or Xcode) made up for them. This gives you a list of files that don't give you a lot of context, at first glance, if you were to execute an `ls` in the directory.

```
ls ~/Library/MobileDevice/Provisioning\ Profiles/
```

Fortunately, you can use the `security` command again to dump the raw info. Pick any one of your `.mobileprovision` files and execute the `security` command, like this:

```
PP_FILE=$(ls ~/Library/MobileDevice/Provisioning\ Profiles/
*mobileprovision | head -1)
security cms -D -i "$PP_FILE"
```

The first command grabs one of the provisioning profiles and assigns it to the `PP_FILE` variable.

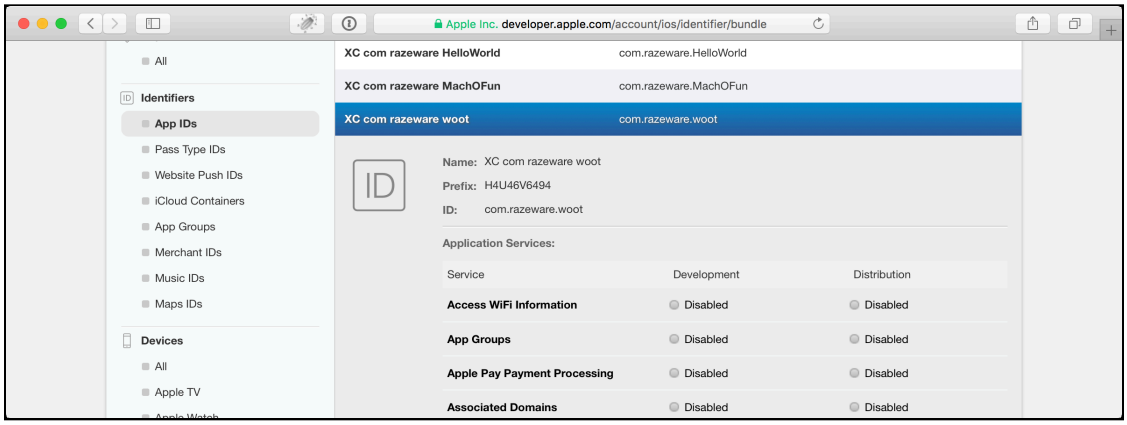
The `PP_FILE` variable is passed into the `security` command which decodes (`-D`) the cryptographic message syntax (`cms`) format of the provisioning profile, specifying the input path via the `-i` option.

The output will be in plist XML form. Your content will be *very* different from mine due to the different nature of the apps you've developed, the entitlements you've specified, the devices and code signatures you've used, as well as the environment you've used (i.e. development/distribution) to generate the provisioning profile.

I'll discuss the output of one of my provisioning profiles as a guide to help explore your own. If you want to follow along word-for-word, my exact provisioning profile is included in the resource directory for this chapter.

From the output, here are some of the highlights:

- **AppIDName** contains the value **XC com razeware woot**. This is the name of the Application ID that is tied to this provisioning profile, which can be found at https://developer.apple.com/account/ios/identifier/bundle.

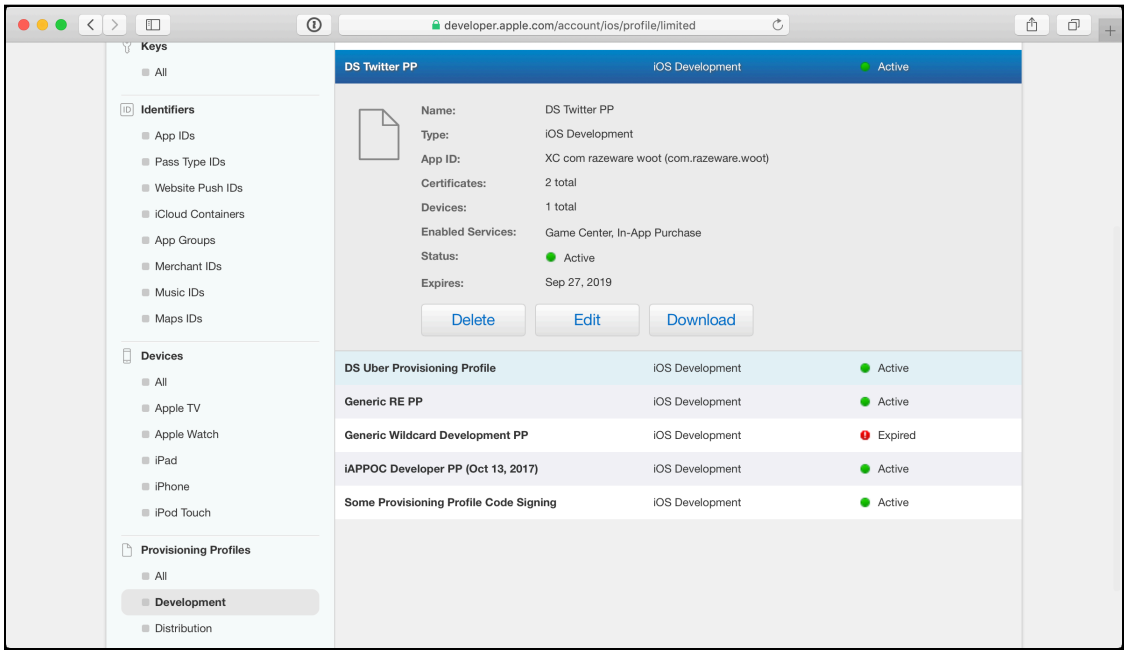


- **TeamIdentifier** contains the value **H4U46V6494**, the unique team ID Apple has given me for my team identity. Apple will generate a specific team ID for every account you pay for. For example, you'll have a unique team ID for App Store builds and a different team ID for Enterprise builds.

- **Entitlements**, unsurprisingly, contains the Entitlements of what the app can and can't do with this signature. This is often the cause of problems in Xcode generated provisioning profiles since Xcode needs to update the App ID configuration (which is essentially the entitlements), and then generate a new provisioning profile with the correct values.

- **IsXcodeManaged** is a Boolean value that indicates if Xcode manages this provisioning profile. The whole code-signing process has caused so many developer headaches that Apple is trying to do more of the work on their end, including signing an app with your distribution certificate. This is a double-edged sword since it's easier to let Xcode manage this for you, but if Xcode does something you didn't expect, the underlying error can be much more difficult to track down.

- **Name** contains the value **DS Twitter PP**, which is the name of the provisioning profile that Apple displays to identify the provisioning profiles on https://developer.apple.com/account/ios/profile/limited.



- **ProvisionedDevices** contains an array of approved devices this provisioning profile can install on, given by a device's UDID.
- **DeveloperCertificates** is an array that contains base64-encoded `x509` certificates. This will contain the same public certificate that was extracted earlier via the `security find-certificate` command. These certificates are also encoded into the actual executables themselves when code signing an application. My provisioning profile contains two different certificates with the exact same name, with one certificate expiring in 2019, and one having already expired in 2018.

Phew! That was a lot of theory, but now you can move on to some actual codesigning work.

Exploring the WordPress app

Just like your typical debugging workflow on your iOS device, before an app is sent up to the Apple iTunes Connect mothership, you must compile an app with a provisioning profile. This provisioning profile is included in every *pre-App Store* `.app` under the name **embedded.mobileprovision**. It's this provisioning profile that tells iOS the application is valid and came from you.

Head over to the resources for this chapter. Then open up the **WordPress.app** container found in the **Pre App Store** directory. If you're using Finder, you can open up the container by right-clicking it and selecting the **Show Package Contents**.

Now head back over to your Terminal window. For the purpose of this tutorial, assign a Terminal variable, **`WORDPRESS`** to the fullpath to the WordPress.app, like so:

```
WORDPRESS="/full/path/to/WordPress.app/"
```

The provisioning profile

Find the **embedded.mobileprovision** provisioning profile inside of the WordPress application and use the **security** command on it.

```
security cms -D -i "$WORDPRESS/embedded.mobileprovision"
```

In this particular provisioning profile, you can see the following:

- Apple has given the **Automattic, Inc.** company the team identifier of **3TMU3BH3NK**.
- The Wordpress app makes use of iCloud services, given the **com.apple.developer.icloud*** keys in the entitlements dictionary. It also looks to make use of certain extension like "App Groups".
- **get-task-allow** is `false`, meaning a debugger can't be attached, as this was app was signed with a distribution signing identity.

Copy the base64-encoded data from the **DeveloperCertificates** key. It should begin with `MIIFozCCBIu...`, and make sure you copy the trailing equals signs if there are any.

Via Terminal, assign this value to a variable named **CERT_DATA**:

```
CERT_DATA=MIIFozCCBIu...
```

The variable `CERT_DATA` now contains the base64-encoded x509 certificate that was used to sign the application.

Now, decode this base64 data and pipe it to **/tmp/wordpress_cert.cer**:

```
echo "$CERT_DATA" | base64 -D > /tmp/wordpress_cert.cer
```

You now have the Wordpress certificate in DER format at **/tmp/wordpress_cert.cer**. You can now execute the following `openssl` command:

```
openssl x509 -in /tmp/wordpress_cert.cer -inform DER -text -noout
```

You'll see the following output:

```
Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number: 786948871528664923 (0xaebcdd447dc4f5b)
    Signature Algorithm: sha256WithRSAEncryption
        Issuer: C=US, O=Apple Inc., OU=Apple Worldwide Developer
Relations, CN=Apple Worldwide Developer Relations Certification Authority
        Validity
            Not Before: Jan 17 13:26:41 2018 GMT
            Not After : Jan 17 13:26:41 2019 GMT
        Subject: UID=PZYM8XX95Q, CN=iPhone Distribution: Automattic, Inc.
(PZYM8XX95Q), OU=PZYM8XX95Q, O=Automattic, Inc., C=US
... truncated ...
```

This means that someone working at "Automattic, Inc" has an identity named "iPhone Distribution: Automattic, Inc. (PZYM8XX95Q)" on their keychain that was used to sign this application.

Embedded executables

Provided an application contains extensions (i.e. share extension, today widgets, or others), there will be even more signed packaged bundles found in the **./Plugins** directory that contain their own application identifier and **embedded.mobileprovision** provisioning profile.

These give the application additional functionality outside the application.

You can verify this on your own time using the same `security` command while drilling into the containers in the `./Plugins` directory, exploring each **embedded.mobileprovision** file respectively.

The _CodeSignature directory

Included in a real iOS application bundle (but not in the Simulator) is a folder named **_CodeSignature** that includes a single file named **CodeResources**. This is an XML plist file which is a checksum of every non-executable file found in this directory.

For example, if you were to execute:

```
cat "$WORDPRESS/_CodeSignature/CodeResources"  | head -10
```

you'll see there is a checksum of value **rSZAWMReahogETtlwDpstztW6Ug=** for the file **AboutViewController.nib**

This can be calculated yourself via `openssl`:

```
openssl sha1 -binary "$WORDPRESS/AboutViewController.nib"  | base64
```

This will produce the matching `rSZAWMReahogETtlwDpsztW6Ug=` value.

Apple has begun the transition from SHA-1 checksums to SHA-256 for iOS applications with Xcode 10 producing checksums for both algorithms.

This `CodeResources` file itself has a checksum performed on the file, which is embedded in the actual WordPress application! This means that if a user were to modify any of the files, or even add a directory in the .app directory without resigning the WordPress app, the iOS application will fail to install on the user's phone.

Resigning the WordPress app

Time for some codesigning fun!

you'll now install the WordPress application onto your iOS device by re-signing the application with your Apple signature.

From a high level standpoint, you'll need to do the following:

1. Copy a valid provisioning profile to the **embedded.mobileprovision** in the WordPress `.app` directory.
2. Change the `Info.plist` key **CFBundleIdentifier** to the new application identifier provided in the new provisioning profile.
3. Re-sign the WordPress application via the identity included in the embedded provisioning profile with the proper entitlements (which is also included in the provisioning profile).

Provided you have a valid, non-expired, provisioning profile that inlcudes your iOS's UDID, you can resign the WordPress app. You can obtain your device's UDID by executing the `mobdevim -f` command when your device is plugged in... (or you can go to iTunes to find it, but that is way less cool).

You can search for valid provisioning profiles at `~/Library/MobileDevice/Provisioning Profiles/` or you can download a valid provisioning profile at https://developer.apple.com/.

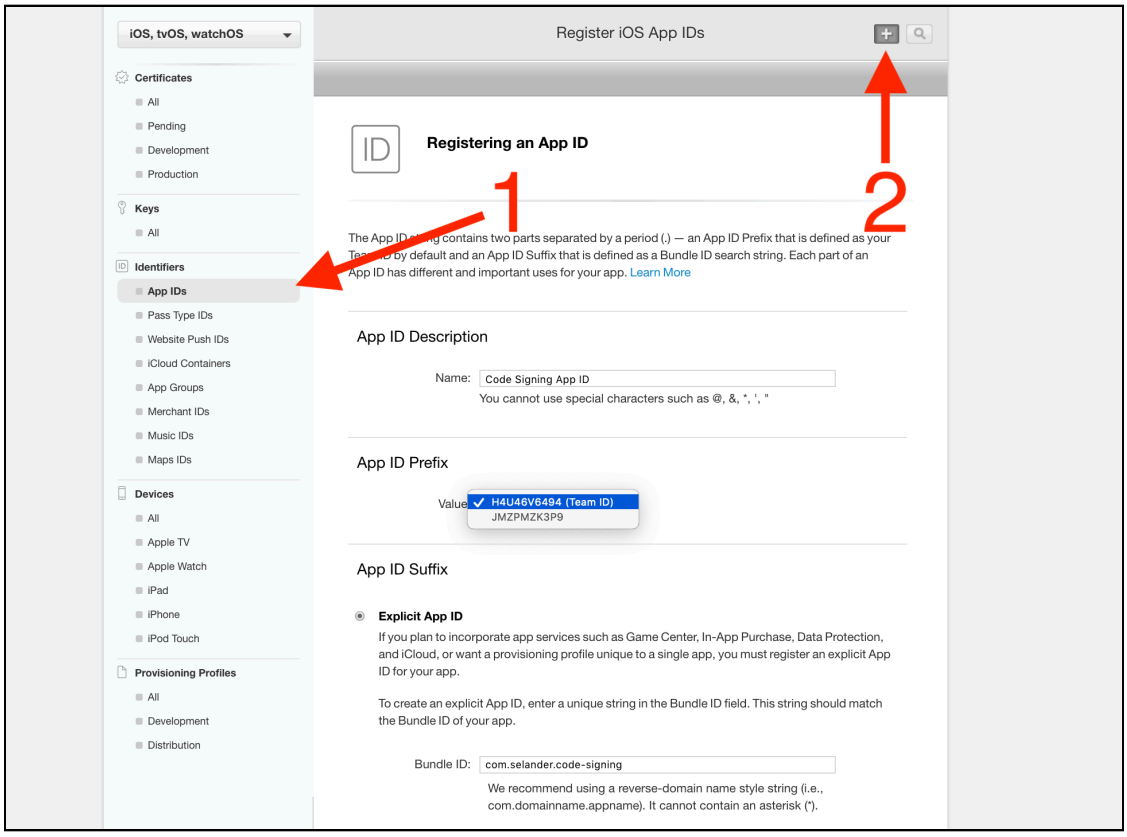
If you have a valid provisioning profile with the above qualifications, you can skip the next step. You can determine if you have a valid provisioning profile by running the same `security cms` command as discussed above.

If you don't have a valid provisioning profile that contains your device and is not expired, you'll need to create a new provisioning profile in the Apple developer portal.

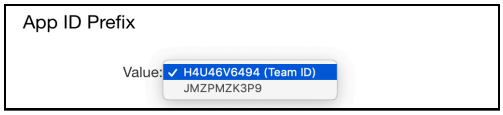
(Optional) Generate a valid provisioning profile

If you don't have a provisioning profile that met the above requirements (UDID, not expired), you'll need to head on over to https://developer.apple.com/ and create a new one.

Although Apple changes the UI/UX on this site from time to time, head on over to the closest equivalent to **Certificates, IDs & Profiles**. Once there, select **App IDs** then create a new App ID via the + button in the upper right corner.



You might be confused if you have multiple App ID prefixes (like me).



To resolve this, remember that everything stems from your signing identity. You can query this information yourself from the commands you performed earlier.

Provided you have a valid `signing identity`, you can use the following Terminal query:

```
security find-certificate -c "iPhone Developer: Derek Selander
(8AW8QLCX5U)" -p > /tmp/public_cert.PEM
```

This extracts the public certificate from the identity. Now you can use `openssl` to search for the App ID prefix which is stored in the **Organizational Unit** (abbreviated as **OU**) in the x509 certificate.

```
openssl x509 -in /tmp/public_cert.PEM -inform pem -noout -text | grep OU=
```

I got the following output:

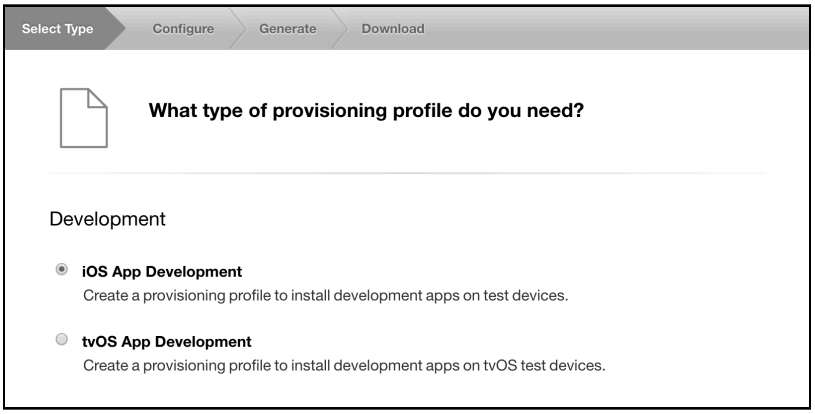
```
Issuer: C=US, O=Apple Inc., OU=Apple Worldwide Developer Relations,
CN=Apple Worldwide Developer Relations Certification Authority

Subject: UID=V969KV7V2B, CN=iPhone Developer: Derek Selander
(8AW8QLCX5U), OU=H4U46V6494, O=Derek Selander, C=US
```

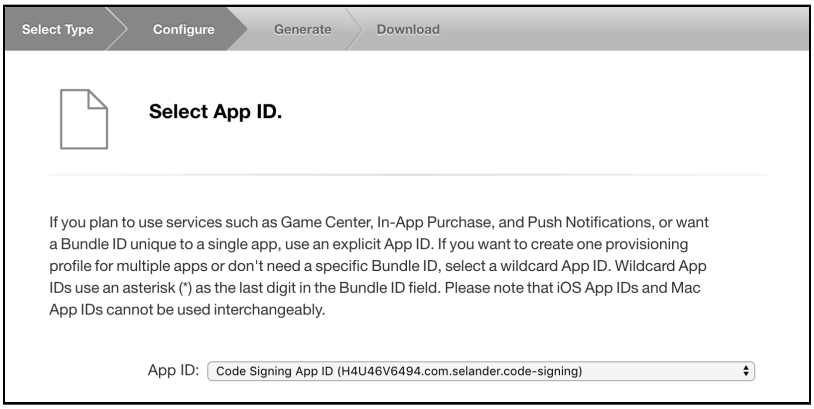
In my case, the signing identity I want to use has the App Prefix **H4U46V6494**, so I'll select that in the Apple Developer portal.

After creating the new App ID in https://developer.apple.com/, head on over to the **Development** section of the **Provisioning Profiles**. Click on the + to add a new provisioning profile.

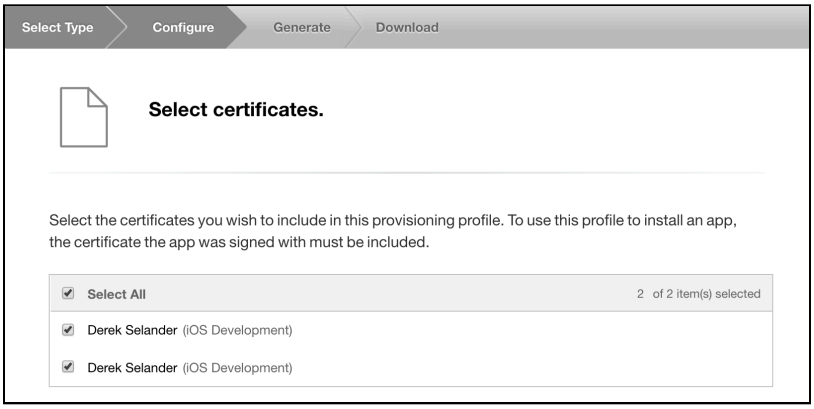
Select **iOS App Development**, the click **Continue**.



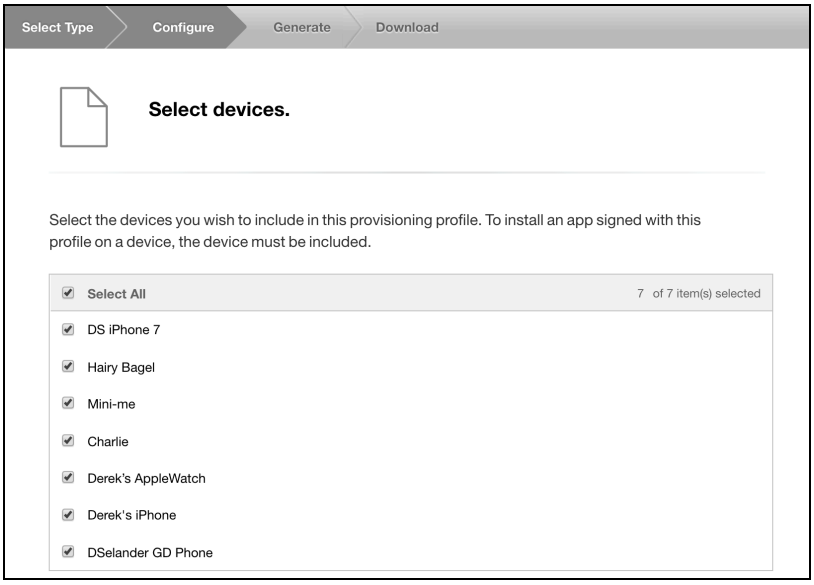
At the next page, select the App ID you just created, then click **Continue** again.



Select all the valid iOS certificates that can be used to sign the iOS application.

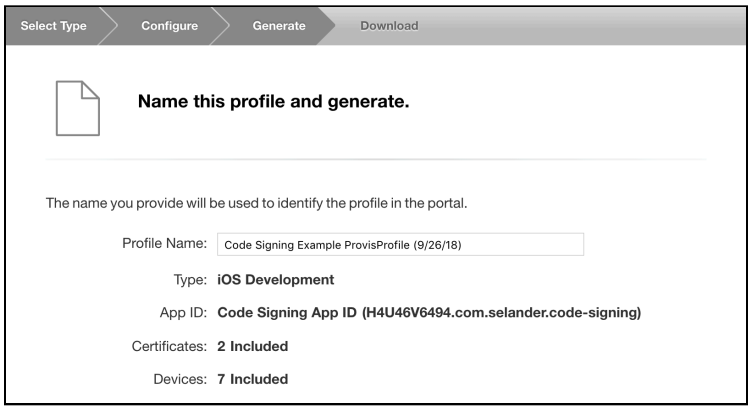


Finally, select all the devices you would like this provisioning profile to be installed on.

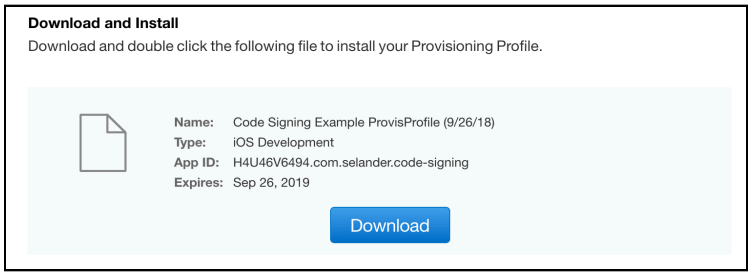


Give the provisioning profile a valid name.

A general word of advice: If you work on a team of iOS developers, and you're generating a Distribution provisioning profile, I would put your initials and date in the name. That way, people know who to track down in case something goes wrong, or if the provisioning profile is expiring.



Once complete, download your newly created provisioning profile. Be sure to save it in a location that you'll remember since you'll be referencing it in a moment.



Copying the provisioning profile

At this point, you should have a valid provisioning profile, which you'll use to resign the WordPress application either by creating a new provisioning profile or by using an existing provisioning profile. Assign the **PP_PATH** Terminal variable to the fullpath of the provisioning profile you expect to use for this experiment.

Your path will be different than mine:

```
PP_PATH=~/Downloads/
Code_Signing_Example_ProvisProfile_92618.mobileprovision
```

Copy the provisioning profile at `PP_PATH` to the `embedded.mobileprovision` file in the WordPress app.

In Terminal, execute:

```
cp "$PP_PATH" "$WORDPRESS/embedded.mobileprovision"
```

Deleting the plugins

The WordPress application has several extension applications embedded into the main app found in the `./Plugins` directory. Each of these contains a unique provisioning profile with a unique application identifier. You could sign each of these extensions itself with a unique provisioning profile, but that will get way too complicated for this demo.

Instead, you'll cripple part of the Wordpress functionality and not use these extensions.

Go ahead and delete the entire `Plugins` directory for the WordPress app.

So now the new, re-signed application will not have functionality for the iOS Today Extension, but that's acceptable for this demo.

Modifying the Info.plist

I hope you've remembered the name of the App ID of the provisioning profile! You'll need to plug that into the Info.plist's key **CFBundleIdentifier** If you don't remember it, you can query it from the provisioning profile.

Here's how to grab that information:

```
security cms -D -i "$PP_PATH" | grep application-identifier -A1
```

This gave me the application identifier I need to plug into the `Info.plist`.

```
<key>application-identifier</key>
<string>H4U46V6494.com.selander.code-signing</string>
```

For me, my application identifier is **H4U46V6494.com.selander.code-signing**.

When you have your application identifier, replace this value in the WordPress's `Info.plist` for the **CFBundleIdentifier** key.

```
plutil -replace CFBundleIdentifier -string H4U46V6494.com.selander.code-
signing "$WORDPRESS/Info.plist"
```

While you're at it, change the display name to further highlight that this is in fact something you can completely tweak to your will. Change around WordPress's visual display name:

```
plutil -replace CFBundleDisplayName -string "Woot" "$WORDPRESS/
Info.plist"
```

This will change around the visual display name to Woot instead of WordPress, provided you can install the application on your iOS device.

Extracting the entitlements

You're almost there!

Your next task is to resign the app with valid entitlements found in the provisioning profile. Since the entitlements get embedded as a dictionary and not as XML in the provisioning profile, it might be easier to extract the entitlements from the main executable first, then patch that file with the new entitlements found in the provisioning profile.

Extract the entitlements to **/tmp/ent.xml**:

```
codesign -d --entitlements :/tmp/ent.xml "$WORDPRESS/WordPress"
```

> **Note**: The above command *appends* to the file — it does not overwrite the file. If you execute this command multiple times, you'll have an incorrectly formatted file, since there will be multiple entitlements at `/tmp/ent.xml`. If you execute this command multiple times, make sure to `rm` the file before executing it again.

Verify the entitlements are valid with a `cat`:

```
cat /tmp/ent.xml
```

Provided the entitlements work, you can extract the entitlements from the current provisioning profile and place them into this new file.

First, write out the provisioning profile XML to a file named **/tmp/scratch**:

```
security cms -D -i "$PP_PATH" > /tmp/scratch
```

Now use the **`xpath`** Terminal command to extract only the entitlement information to the clipboard.

By the way, you should play around with this command first before piping it to the clipboard (with `pbcopy`) so you understand the content it's grabbing.

```
xpath /tmp/scratch '//*[text() = "Entitlements"]/following-sibling::dict'
| pbcopy
```

You now have the valid entitlements in your clipboard. Open up **/tmp/ent.xml**, remove the enclosing `<dict>`'s contents and replace with the contents of your clipboard.

Your finalized `/tmp/ent.xml` file should look like the following entitlements:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN" "http://
www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
    <key>keychain-access-groups</key>
    <array>
        <string>H4U46V6494.*</string>
    </array>
    <key>get-task-allow</key>
    <true />
    <key>application-identifier</key>
    <string>H4U46V6494.com.selander.code-signing</string>
    <key>com.apple.developer.team-identifier</key>
    <string>H4U46V6494</string>
</dict>
</plist>
```

This file is also included in the chapter directory in case you want to start with that instead.

Finally, signing the WordPress app

You now have performed all the setup. You have a valid signing identity; you have a valid provisioning profile embedded in the WordPress application at `embedded.mobileprovision`; you have removed the `Plugins` directory; and you have the entitlements of the new provisioning profile found at **/tmp/ent.xml**.

You can now sign the application with your signing identity!

Before you do that, make a duplicate backup of the WordPress app, because it's easy to screw this part up, and it's tricky to undo the action if you do screw up.

Once you have a duplicate of your WordPress application, use the **`codesign`** command with your signing identity on the WordPress applications **Frameworks** directory:

```
codesign -f -s "iPhone Developer: Derek Selander (8AW8QLCX5U)"
"$WORDPRESS"/Frameworks/*
```

You need to sign this directory first.

```
codesign --entitlements /tmp/ent.xml -f -s "iPhone Developer: Derek
Selander (8AW8QLCX5U)" "$WORDPRESS"
```

Now for the moment of truth! See if you can install the WordPress application. In Terminal, type:

```
mobdevim -i "$WORDPRESS"
```

Did it succeed?

Provided you have followed the steps *exactly*, you'll see a new app with the WordPress logo with the name "Woot" underneath it!

Even better, provided you signed your application with a developer provisioning profile, you'll have the `get-task-allow` entitlement, meaning you can debug this WordPress application!

Launch the newly installed WordPress application on your iOS device.

Fire up Xcode, select the **Debug** menu, select **Attach to Process** and search for the WordPress application.

Alternatively, you can also use `mobdevim` to debug your application without having to use Xcode.

Simply type the following:

```
mobdevim -d $WORDPRESS
```

This will set up LLDB to the state just before launch. Prove to yourself that you have complete control over this WordPress process by forcing WordPress to launch in Spanish, via specifying the iOS language environment variable.

```
(lldb) run -AppleLanguages "(es)"
```



Did it fail?

If the output says "success", then you're good to go. If not, open a new Terminal window and execute the following:

```
mobdevim -c | grep installd
```

This will dump out all logs pertaining to the daemon `installd`. This daemon is responsible for installing your application and will provide useful logging information as to why your application failed to install. From there, you'll need to careful review the above steps and repeat the process.

Once the log monitoring is running, repeat the above `mobdevim -i "$WORDPRESS"` command so you can capture the installation error.

If you can't get it to work, I've included a shell script that will resign the WordPress application for you. It's named **dsresign** and expects the path to the application as argument one, followed by the path to the provisioning profile you want to resign the application with.

Where to go from here?

This chapter has only scratched the surface of code signing. There is a lot more great content out there that focuses on other components to code signing.

If you want to truly know the in and outs of code signing, check out Jonathan Levin's *OS Internals Volume III Security & Insecurity*, which discusses code signing at an unprecedented level and gives you a look at everything, right down to the C structs.

Also check out this article, which is one of the my favorite code signing articles out there from a developer standpoint:

- https://www.objc.io/issues/17-security/inside-code-signing/

Section IV: Custom LLDB Commands

You've learned the basic LLDB commands, the assembly that goes into code and the miscellaneous low-level concepts that make a program…well, a program.

It's time to put that knowledge together to create some very powerful and complex debugging scripts. As you will soon see, you're only limited by your skill and imagination — and finding the correct class (or header file) to do your debugging bidding.

LLDB ships with an integrated Python module that allows you to access most parts of the debugger through Python. This lets you leverage all the power of Python (and its modules) to help uncover whatever dark secrets vex you.

Chapter 21: Hello Script Bridging

Chapter 22: Debugging Script Bridging

Chapter 23: Script Bridging Classes & Hierarchy

Chapter 24: Script Bridging with Options & Arguments

Chapter 25. Script Bridging with SBValue & Language Contexts

Chapter 26. SB Examples, Improved Lookup

Chapter 27. SB Examples, Resymbolicating a Stripped ObjC Binary

Chapter 28. SB Examples, Malloc Logging

Chapter 21: Hello, Script Bridging

LLDB has several ways you can use to create your own customized commands. The first way is through the easy-to-use `command alias` you saw in Chapter 9, "Persisting and Customizing". This command simply creates an alias for a static command. While easy to implement, it really only allowed you to execute commands with no input.

After that came the `command regex`, which let you specify a regular expression to capture input then apply it to a command. You learned about this command in Chapter 10, "Regex Commands". This command works well when you want to feed input to an LLDB command, but it was inconvenient to execute multiline commands and supplying multiple, optional parameters could get really messy.

Next up in the tradeoff between convenience and complexity is LLDB's **script bridging**. With script bridging, you can do nearly anything you like. Script bridging is a **Python** interface LLDB uses to help extend the debugger to accomplish your wildest debugging dreams.

However, there's a cost to the script bridging interface. It has a steep learning curve, and the documentation, to put it professionally, sucks. Fortunately, you've got this book in your hands to help guide you through learning script bridging. Once you've a grasp on LLDB's Python module, you can do some very cool (and very scary!) things.

Credit where credit's due

Before we officially begin talking about script bridging, I want to bring up one Python script that has blown my mind. If it wasn't for this script, this book would not be in your hands.

```
/Applications/Xcode.app/Contents/SharedFrameworks/LLDB.framework/
Versions/A/Resources/Python/lldb/macosx/heap.py
```

This is *the* script that made me take a deep dive into learning LLDB. I've never had a mental butt-kicking as good as I did trying to initially understand what was happening in this code.

This script had it all: finding stack traces for `malloc`'d objects (`malloc_info -s`), getting all instances of a particular subclass of `NSObject` (`obj_refs -O`), finding all pointers to a particular reference in memory (`ptr_refs`), finding C strings in memory (`cstr_ref`).

You can load the contents of this script with the following LLDB command:

```
(lldb) command script import lldb.macosx.heap
```

Sadly, this script has fallen a bit out of functionality as the compiler has changed, while this code has not, rendering several of its components unusable.

When you're done reading this section, I would strongly encourage you to attempt to understand the contents of this script. You can learn *a lot* from it.

Ok, now back to our regularly scheduled, reading program…

Python 101

As mentioned, LLDB's script bridge is a Python interface to the debugger. This means you can load and execute Python scripts in LLDB. In those Python scripts, you include the `lldb` module to interface with the debugger to obtain information such as the arguments to a custom command.

Don't know Python? Don't fret. Python is one of the most friendly languages to learn. And just like the Swift Playgrounds everyone's losing their mind over, Python has an attractive REPL for learning.

> **Note**: At the time of writing, there are signs LLDB is slowly migrating over from Python version 2 to Python 3. Just like Swift, there are breaking changes in these different versions. In order to make sure you're learning the correct version of Python, you need to know which version of Python LLDB is using. At the time of writing, LLDB uses Python 2.7.10.

Let's figure out which version of Python LLDB is using. Open a Terminal window and type the following:

```
lldb
```

As expected, LLDB will start. From there, execute the following commands to find out which Python version is linked to LLDB:

```
(lldb) script import sys
(lldb) script print (sys.version)
```

The **`script`** command brings up the Python interpreter for LLDB. If you just typed in `script` without arguments, you'd be greeted with LLDB's Python REPL.

If LLDB's Python version is different than 2.7.x, freak out and complain loudly on the book's forum.

> **Note**: If the version is 2.7.x this is still valid. As long as the version you're running is not Python 3.X.Y by default your system will work as described. Your system-installed Python version does not have to match 2.7.10 exactly; bug fix releases work fine also.

Now you know the Python version LLDB works with, ensure you have the correct version of Python symlinked to the `python` Terminal command. Open a new Terminal window and type the following:

```
python --version
```

If the Python version matches the one that LLDB has, then launch Python with no arguments in the Terminal:

```
python
```

If you have a different version of Python symlinked (i.e. 3.X.Y), you need to launch Python with the correct version number. For example, in Terminal, type `python` and press Tab. Different version(s) of Python might pop up with the correct version number.

Enter the correct version number associated with the LLDB version of Python:

```
python2.7
```

Either way, ensure the LLDB version of Python matches the one you have in your Terminal:

```
>>> import sys
>>> print (sys.version)
```

Notice in the actual Python REPL there's no need to prefix any of the commands with the LLDB script command.

Playing around in Python

If you are unfamiliar with Python, this section will help you quickly get familiar with the language. If you're already knowledgeable about Python, feel free to jump to the next section.

In your Terminal session, open a Python REPL by typing the following:

```
python
```

Next, in the Python REPL, type the following:

```
>>> h = "hello world"
>>> h
```

You'll see the following output:

```
'hello world'
```

Python lets you assign variables without needing to declare the type beforehand. Unlike Swift, Python doesn't really have the notion of constants, so there's no need for a `var` or `let` declaration for a variable.

> **Note**: If you have a different version of Python, then some of the commands might have different syntax. You'll need to consult Google to figure out the correct equivalent command.

Going a step further, play around with the variable `h` and do some basic string manipulation:

```
>>> h.split(" ")
['hello', 'world']
```

This will give a Python **list**, which is somewhat like an array that can store different types of objects.

If you need your Swift fix equivalent, then imagine a list is something similar to the following Swift code:

```
var h: [Any] = []
```

You can verify this by looking up the Python's class type. In Terminal, press the up arrow to bring up the previous command and append the `.__class__` call to the end like so:

```
>>> h.split(" ").__class__
<type 'list'>
```

Note there's two underscores preceding and following the word *class*.

What type of class is the `h` variable?

```
>>> h.__class__
<type 'str'>
```

That's good to know; a string is called `str`. You can get help on the `str` object by typing the following:

```
>>> help (str)
```

This will dump all the info pertaining to `str`, which is too much to digest at the moment.

Exit out of this documentation by typing the `q` character and narrow your search by looking only for the `split` function used previously:

```
>>> help (str.split)
```

You'll get some documentation output similar to the following:

```
Help on method_descriptor:

split(...)
    S.split([sep [,maxsplit]]) -> list of strings

    Return a list of the words in the string S, using sep as the
    delimiter string.  If maxsplit is given, at most maxsplit
    splits are done. If sep is not specified or is None, any
    whitespace string is a separator and empty strings are
    removed from the result.
```

Reading the above documentation, you can see the first optional argument expects a string, and an optional second argument to indicate the maximum upper limit to split the string.

What do you think will happen when you try to execute the following command? Try your best to figure it out before executing it.

```
>>> h.split(" ", 0)
```

Now to turn your attention towards functions. Python uses indentation to define scope, instead of the braces that many other languages use, including Swift and Objective-C. This is a nice feature of Python, since it forces developers to not be lazy slobs with their code indentation.

Declare a function in the REPL:

```
>>> def test(a):
...
```

You'll get an ellipsis as output, which indicates you have started creating a function. Type two spaces and then enter the following code. If you don't have a consistent indentation, the python function will produce an error.

```
...   print(a + " world!")
```

Press Enter again to exit out of the function. Now, test out your newly created `test` function:

```
>>> test("hello")
```

You'll get the expected `hello world!` printed out.

Now that you can "truthfully" put three years of Python experience on your resume, it's time to create an LLDB Python script.

Creating your first LLDB Python script

From here on out, you'll be creating all your LLDB Python scripts in the `~/lldb` directory. If you want to have them in a different directory, everytime I say `~/lldb`, you'll need to invoke your "mental symlink" to whatever directory you've decided to use.

In Terminal, create the ~/lldb directory:

```
mkdir ~/lldb
```

In your favorite ASCII text editor, create a new file named **helloworld.py** in your newly created `~/lldb` directory. For this particular example, I'll use the my-editor-is-better-neutral-argument, `nano`.

```
nano ~/lldb/helloworld.py
```

Add the following code to the file:

```
def your_first_command(debugger, command, result, internal_dict):
  print ("hello world!")
```

Make sure you indent the `print ("hello world")` line (ideally with two spaces) or else it won't be included as part of the function!

For now, ignore the parameters passed into the function. Remember when you learned about your `hello_world.c` or `hello_world.java`, and the instructor (or the internet) said to just ignore the params in main for now? Yeah, same thing here. These params are the defined way LLDB interacts with your Python code. You'll explore them in upcoming chapters.

Save the file. If you're using `nano`, `Ctrl + O` will write to disk.

Create a new tab in Terminal and launch a new LLDB session:

```
lldb
```

This will launch a blank, unattached LLDB session.

In this new LLDB session, import the script you created:

```
(lldb) command script import ~/lldb/helloworld.py
```

If the script is imported successfully, there will be no output.

But how do you execute the command? The only thing the above command did was bring the `helloworld` (yes, named after the file) module's path in as a candidate to use for Python.

If you plan to use this function in Python, you'll need to import the module if you want to use any of the functions. Type the following into LLDB:

```
(lldb) script import helloworld
```

You can verify you've successfully imported the module by dumping all the methods in the `helloworld` python module:

```
(lldb) script dir(helloworld)
```

The `dir` function will dump the contents of the module. If you successfully imported the module, you'll see the following output:

```
['__builtins__', '__doc__', '__file__', '__name__', '__package__',
'your_first_command']
```

Take note, the function you created earlier: `your_first_command` is listed in the output.

Although the above two commands weren't necessary to set up the command, it does show you how this script bridging works. You imported the `helloworld` module into the Python context of LLDB, but when you execute normal commands, you aren't executing in a Python context (although the command logic underneath could be using Python).

So how do you make your command available only through LLDB, and not through the Python context of LLDB?

Head back to LLDB and type the following:

```
(lldb) command script add -f helloworld.your_first_command yay
```

This adds a command to LLDB, which is implemented in the `helloworld` Python module with the function `your_first_command`. This scripted function is assigned to the LLDB command **`yay`**.

Execute the `yay` command now:

```
(lldb) yay
```

Provided everything worked, you'll get the expected `hello world!` output.

Setting up commands efficiently

Once the high of creating a custom function in script bridging has worn off, you'll come to realize you don't want to type this stuff each time you start LLDB. You want those commands to be there ready for you as soon as LLDB starts.

Fortunately, LLDB has a lovely function named **`__lldb_init_module`**, which is a hook function called as soon as your module loads into LLDB.

This means you can stick your logic for creating the LLDB command in this function, eliminating the need to manually set up your LLDB function once LLDB starts!

Open the `helloworld.py` class you created and add the following function below `your_first_command`'s definition:

```
def __lldb_init_module(debugger, internal_dict):
    debugger.HandleCommand('command script add -f
helloworld.your_first_command yay')
```

Here you're using a parameter passed into the function named `debugger`. With this object, an instance of `SBDebugger`, you're using a method available to it called `HandleCommand`. Calling `debugger.HandleCommand` is pretty much equivalent to typing something into LLDB.

For example, if you typed: `po "hello world"`, the equivalent command would be `debugger.HandleCommand('po "hello world"')`

Remember the python `help` command you used earlier? You can get help documentation from this command by typing:

```
(lldb) script help(lldb.SBDebugger.HandleCommand)
```

At the time of writing, you'll get a rather disappointing amount of help documentation:

```
HandleCommand(self, *args) unbound lldb.SBDebugger method
    HandleCommand(self, str command)
```

That's why there's such a steep learning curve to this stuff, and the reason not many people venture into learning about script bridging. That's why you picked up this book, right?

Save your **helloworld.py** file and open up your `~/.lldbinit` file in your favorite editor.

You're now going to specify you want the `helloworld` module to load at startup every time LLDB loads up.

At the end of the file, add the following line:

```
command script import ~/lldb/helloworld.py
```

Save and close the file.

Open Terminal and start up another tab with LLDB in it like so:

```
lldb
```

Since you specified to have the `helloworld` module imported into LLDB upon startup, and you also specified to create the `yay` function as soon as the `helloworld` python module loads through the `__lldb_init_module` module, the `yay` LLDB command will be available immediately to you.

Try it out now:

```
(lldb) yay
```

If everything went well you'll see the following output:

```
hello world!
```

Awesome! You now have a foundation for building some very complex scripts into LLDB. In the following chapters, you'll explore more of how to use this incredibly powerful tool.

For now, close all those Terminal tabs and give yourself a pat on the back.

Where to go from here?

If you don't feel comfortable with Python, now is the time to start brushing up on it. If you have past development experience, you'll find Python to be a fun and friendly language to learn. It's a great language for quickly building other tools to help with everyday programming tasks.

Chapter 22: Debugging Script Bridging

You've learned the basics of LLDB's Python script bridging. Now you're about to embark on the frustrating yet exhilarating world of making full LLDB Python scripts.

As you learn about the classes and methods in the Python lldb module, you're bound to make false assumptions or simply type incorrect code. *In short, you're going to screw up*. Depending on the error, sometimes these scripts fail silently, or they may blow up with an angry `stderr`.

You need a methodical way to figure out what went wrong in your LLDB script so you don't pull your hair out. In this chapter, you'll explore how to inspect your LLDB Python scripts using the Python `pdb` module, which is used for debugging Python scripts. In addition, you can execute your own "normal" Objective-C, Objective-C++, C or Swift code (or even other languages) within **SBDebugger**'s (or **SBCommandReturnObject**'s) **HandleCommand** method.

In fact, there's alternative ways to execute non-Python code that you'll learn about in an upcoming chapter, but for now, you'll stick to **HandleCommand** and see how to manage a build time error, or fix a script that produces an incorrect result.

Although it might not seem like it at first, this is the **most important** chapter in the LLDB Python section, since it will teach you how to explore and debug methods while you're learning this new Python module. I would have (figuratively?) killed for a chapter like this when I was first learning the **Script Bridging** module.

Debugging your debugging scripts with pdb

Included in the Python distribution on your system is a Python module named `pdb` you can use to set breakpoints in a Python script, just like you do with LLDB itself! In addition, `pdb` has other debugging essential features that let you step into, out of, and over code to inspect potential areas of interest.

You're going to continue using the **helloworld.py** script in **~/lldb** from the previous chapter. If you haven't read that chapter yet, copy the **helloworld.py** from the **starter** directory into a directory named **lldb** inside your home directory.

Either way, you should now have a file at **~/lldb/helloworld.py**.

Open up **helloworld.py** and navigate to the `your_first_command` function, replacing it with the following:

```
def your_first_command(debugger, command, result, internal_dict):
    import pdb; pdb.set_trace()
    print ("hello world")
```

> **Note**: It's worth pointing out `pdb` will not work when you're debugging Python scripts in Xcode. The Xcode console window will hang once `pdb` is tracing a script, so you'll need to do all pdb Python script debugging in a Terminal window.

Save your changes and open a Terminal window to create a new LLDB session. In Terminal, type:

```
lldb
```

Next, execute the `yay` command (which is defined in `helloworld.py`, remember?) like so:

```
(lldb) yay woot
```

Execution will stop and you'll get output similar to the following:

```
> /Users/derekselander/lldb/helloworld.py(3)your_first_command()
-> print ("hello world")
(Pdb)
```

The LLDB script gave way to `pdb`. The Python debugger has stopped execution on the `print` line of code within `helloworld.py` inside the function `your_first_command`.

When creating a LLDB command using Python, there are specific parameters expected in the defining Python function. You'll now explore these parameters, namely **`debugger`**, **`command`**, and **`result`**.

Explore the `command` argument first, by typing the following into your `pdb` session:

```
(Pdb) command
```

This will dump out the commands you supplied to your `yay` custom LLDB command. This will always come in the form of a `str`, even if you have multiple arguments or integers as input. Since there's no logic to handle any commands, the `yay` command will silently ignore all input. If you typed in `yay woot` as indicated earlier, only `woot` would be spat out as the `command`.

Next up on the parameter exploration list is the `result` parameter. Type the following into `pdb`:

```
(Pdb) result
```

This will dump out something similar to the following:

```
<lldb.SBCommandReturnObject; proxy of <Swig Object of type
'lldb::SBCommandReturnObject *' at 0x110323060> >
```

This is an instance of `SBCommandReturnObject`, which is a class the lldb module uses to let you indicate if the execution of an LLDB command was successful. In addition, you can append messages that will be displayed when your command finishes.

Type the following into `pdb`:

```
(Pdb) result.AppendMessage("2nd hello world!")
```

This appends a message which will be shown by LLDB when this command finishes. In this case, once your command finishes executing, `2nd hello world!` will be displayed. However, your script is still frozen in time thanks to `pdb`.

Once your LLDB scripts get more complicated, the `SBCommandReturnObject` will come into play, but for simple LLDB scripts, it's not really needed. You'll explore the `SBCommandReturnObject` command more later in this chapter.

Finally, onto the debugger parameter. Type the following into `pdb`:

```
(Pdb) debugger
```

This will dump out another object of class `SBDebugger`, similar to the following:

```
<lldb.SBDebugger; proxy of <Swig Object of type 'lldb::SBDebugger *' at
0x110067180> >
```

You explored this class briefly in the previous chapter to help create the LLDB `yay` command. You've already learned one of the most useful commands in `SBDebugger`: `HandleCommand`.

Resume execution in `pdb`. Like LLDB, it has logic to handle a `c` or `continue` to resume execution.

Type the following into `pdb`:

```
(Pdb) c
```

You'll get the following output:

```
hello world!
2nd hello world!
```

`pdb` is great when you need to pause execution in a certain spot to figure out what's gone wrong. For example, you could have some complicated setup code, and pause in an area where the logic doesn't seem to be correct.

This is a much more attractive solution than constantly typing `script` in LLDB to execute one line of Python code at a time.

pdb's post mortem debugging

Now that you've a basic understanding of the process of debugging your scripts, it's time to throw you into the deep end with an actual LLDB script and see if you can fix it using `pdb`'s post-mortem debugging features.

Depending on the type of error, `pdb` has an attractive option that lets you explore the problematic stack trace in the event the code you're running threw an exception. This type of debugging methodology will only work if Python threw an exception; this method will *not* work if you receive unexpected output but your code executed without errors.

However, if your code has error handling (and as your scripts get more complex, they really should), you can easily hunt down potential errors while building your scripts.

Find the **starter** folder of the resources for this chapter. Next, copy the `findclass.py` file over to your default `~/lldb` directory. Remember, if you're stubborn and decided to go with a different directory location, you'll need to adjust accordingly.

Don't even look at what this code does yet. It's not going to finish executing as-is, and you'll use `pdb` to inspect it after you view the error.

Once the script has been copied to the correct directory, open a Terminal window and launch and attach LLDB to any program which contains Objective-C. You could choose a macOS application or something on the iOS Simulator, or maybe even a watchOS application.

For this example, I'll attach to the macOS `Photos` application, but you're strongly encouraged to attach to a different application. Hey, that's part of being an explorer!

> **Note**: You will need to disable SIP to attach to any process on your mac. Also in LLDB version 1000.11.37.1, there's a pretty serious bug which incorrectly imports the macOS headers — even if you are attached to an iOS Simulator application. You can see if this bug affects you by executing a `po @import Foundation` in LLDB and observing the output. If you're affected by this bug, you will need to use a different version of LLDB or check out the Appendix to get around this bug.

Make sure the application is alive and running and attach LLDB to it:

```
lldb -n Photos
```

Once the process has attached, import the new script into LLDB:

```
(lldb) command script import ~/lldb/findclass.py
```

Provided you placed the script in the correct directory, you should get no output. The script will install quietly.

Figure out what this command does by looking at the documentation, since you haven't even looked at the source code for it yet. Type the following into LLDB:

```
(lldb) help findclass
```

You'll get output similar to the following:

```
Syntax: findclass

The `findclass` command will dump all the Objective-C runtime classes it
knows about. Alternatively, if you supply an argument for it, it will do
a case-sensitive search looking only for the classes that contain the
input.
```

```
Usage: findclass  # All Classes
Usage: findclass UIViewController # Only classes that contain
UIViewController in name
```

Cool! Let's try this command. Try dumping out all classes the Objective-C runtime knows about.

```
(lldb) findclass
```

You'll get a rather annoying error assertion similar to the following:

```
Traceback (most recent call last):
  File "/Users/derekselander/lldb/findclass.py", line 40, in findclass
    raise AssertionError("Uhoh... something went wrong, can you figure it
out? :]")
AssertionError: Uhoh... something went wrong, can you figure it out? :]
```

It's clear the author of this script is horrible at providing decent information into what happened in the AssertionError. Fortunately, it raised an error! You can use `pdb` to inspect the stack trace at the time the error was thrown.

In LLDB, type the following:

```
(lldb) script import pdb
(lldb) findclass
(lldb) script pdb.pm()
```

This imports `pdb` into LLDB's Python context, runs `findclass` again, then asks `pdb` to perform a "post mortem".

LLDB will change to the pdb interface and jump to the line that threw the error.

```
> /Users/derekselander/lldb/findclass.py(40)findclass()
-> raise AssertionError("Uhoh... something went wrong, can you figure it
out? :]")
(Pdb)
```

From here, you can use `pdb` as your new BFF to help explore what's happening.

Speaking of what's happening, you haven't even looked at the source code yet! Lets change that. Type the following into `pdb`:

```
(Pdb) l 1, 50
```

This will list lines 1, 50 of the `findclass.py` script.

You have the typical function signature which handles the majority of the logic in these commands:

```
def findclass(debugger, command, result, internal_dict):
```

Next up in interesting tidbits is a big long string named **codeString**, which starts its definition on line 18. It's a Python multi-line string, which starts with three quotes and finishes with three quotes on line 35. This string is where the meat of this command's logic lives.

In your pdb session, type the following:

```
(Pdb) codeString
```

You'll get some not-so-pretty output, since dumping a Python string includes all newlines.

```
'\n    @import Foundation;\n    int numClasses;\n    Class * classes =
NULL;\n    classes = NULL;\n    numClasses = objc_getClassList(NULL, 0);
\n    NSMutableString *returnString = [NSMutableString string];\n
classes = (__unsafe_unretained Class *)malloc(sizeof(Class) *
numClasses);\n    numClasses = objc_getClassList(classes, numClasses);
\n\n    for (int i = 0; i < numClasses; i++) {\n      Class c =
classes[i];\n      [returnString appendFormat:@"%s,", class_getName(c)];
\n    }\n    free(classes);\n    \n    returnString;\n    '
```

Let's try that again. Use pdb to print out a pretty version of the codeString variable.

```
(Pdb) print codeString
```

Much better!

```
@import Foundation;
int numClasses;
Class * classes = NULL;
classes = NULL;
numClasses = objc_getClassList(NULL, 0);
NSMutableString *returnString = [NSMutableString string];
classes = (__unsafe_unretained Class *)malloc(sizeof(Class) *
numClasses);
numClasses = objc_getClassList(classes, numClasses);

for (int i = 0; i < numClasses; i++) {
  Class c = classes[i];
  [returnString appendFormat:@"%s,", class_getName(c)];
}
free(classes);

returnString;
```

This codeString contains Objective-C code which uses the Objective-C runtime to get all the classes it knows about. The final line of this code, returnString, essentially lets you return the value of returnString back to the Python script. More on that shortly.

Scan for the next interesting part. On line 40, the debugger is currently at a `raise` call. This is also the line that provided the annoyingly vague message you received from LLDB.

```
37    res = lldb.SBCommandReturnObject()
38    debugger.GetCommandInterpreter().HandleCommand("po " ...
39    if res.GetError():
40 ->     raise AssertionError("Uhoh... something went wron...
41    elif not res.HasResult():
42        raise AssertionError("There's no result. Womp wom...
```

Note the –> on line 40. This indicates where `pdb` is currently paused.

But wait, `res.GetError()` looks interesting. Since everything is fair game to explore while `pdb` has the stack trace, why don't you explore this error to see if you can actually get some useful info out of this?

```
(Pdb) print res.GetError()
```

There you go! Depending whether you decided to break on a macOS, iOS, watchOS, or tvOS app, you might get a slightly different count of error messages, but the idea is the same.

```
error: warning: got name from symbols: classes
error: 'objc_getClassList' has unknown return type; cast the call to its
declared return type
error: 'objc_getClassList' has unknown return type; cast the call to its
declared return type
error: 'class_getName' has unknown return type; cast the call to its
declared return type
```

The problem here is the code within `codeString` is causing LLDB some confusion. This sort of error is very common in LLDB. You often need to tell LLDB the return type of a function, because it doesn't know what it is. In this case, both `objc_getClassList` and `class_getName` have unknown return types.

A quick consultation with Google tells us the two problematic methods in question have the following signatures:

```
int objc_getClassList(Class *buffer, int bufferCount);
const char * class_getName(Class cls);
```

All you need to do is cast the return type to the correct value in the `codeString` code.

Open up `~/lldb/findclass.py` and replace the definition of `codeString` with the following:

```
codeString = r'''
@import Foundation;
```

```
  int numClasses;
  Class * classes = NULL;
  classes = NULL;
  numClasses = (int)objc_getClassList(NULL, 0);
  NSMutableString *returnString = [NSMutableString string];
  classes = (__unsafe_unretained Class *)malloc(sizeof(Class) *
  numClasses);
  numClasses = (int)objc_getClassList(classes, numClasses);

  for (int i = 0; i < numClasses; i++) {
    Class c = classes[i];
    [returnString appendFormat:@"%s,", (char *)class_getName(c)];
  }
  free(classes);

  returnString;
  '''
```

Save your work and jump back to your LLDB Terminal window. You'll still be inside `pdb`, so type `Ctrl + D` to exit. Next, type the following:

```
(lldb) command script import ~/lldb/findclass.py
```

This will reload the script into LLDB with the new changes in the source code. This is required if you make any changes to the source code and you want to test out the command again without having to restart LLDB.

Try your luck again and dump all of the Objective-C classes available in your process.

```
(lldb) findclass
```

Boom! You'll get a slew of output containing all the Objective-C classes in your program. From your app, from `Foundation`, from `CoreFoundation`, and so on. Heh… there's more than you thought there would be, right?

Try limiting your query to something slightly more manageable. Search for all classes containing the word `ViewController`:

```
(lldb) findclass ViewController
```

Depending on the process you've attached to, you'll get a different amount of classes containing the name ViewController.

When developing commands using the Python script bridging, `pdb` is a superb tool to keep in your toolbox to help you understand what is happening. It works well for inspecting complicated sections and breaking on problematic areas in your Python script.

expression's Debug Option

As you saw in Chapter 5, "Expression," LLDB's `expression` command has a slew of options available for when LLDB is evaluating code provided to this command. One of these options, overlooked until now, is the **`--debug`** option, or more simply **`-g`**. If you supply this option to `expression`, LLDB will evaluate the expression, but the expression will be written to a file and control will stop as soon as execution hits your command.

Confused? Maybe it would be better to see this option in action. Jump back to your **findclass.py** file and jump to line 38, which contains the following line of code:

```
debugger.GetCommandInterpreter().HandleCommand("expression -lobjc -O -- "
+ codeString, res)
```

In the options section of the `expression` command, add the `-g` option so it now looks like the following:

```
debugger.GetCommandInterpreter().HandleCommand("expression -lobjc -g -O
-- " + codeString, res)
```

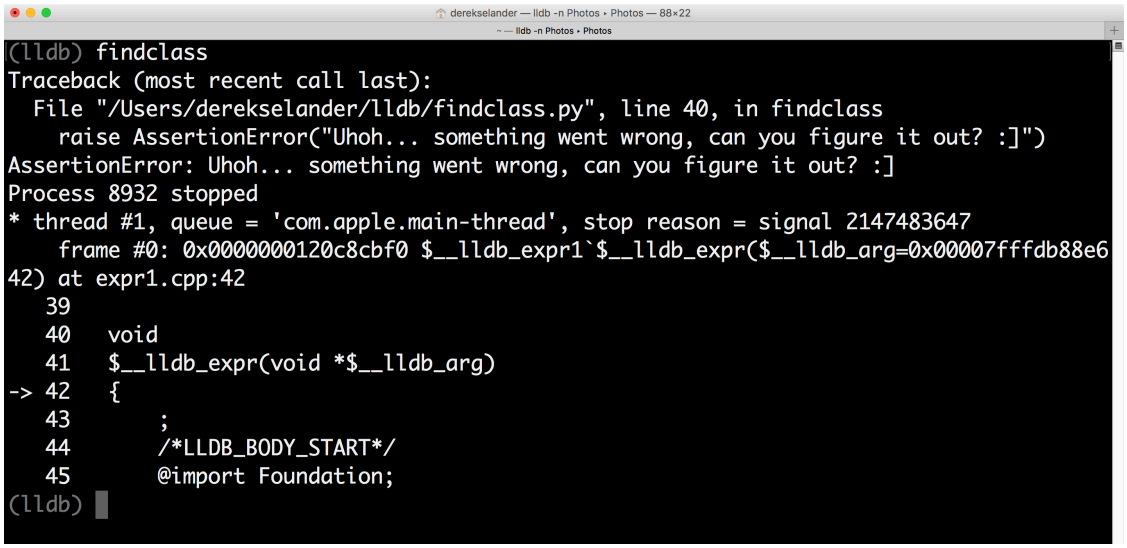
Save your work in **findclass.py** and reload your script through LLDB:

```
(lldb) command script import ~/lldb/findclass.py
```

Once reloaded, give `findclass` a spin:

```
(lldb) findclass
```

Execution will now stop in a method created by the JIT (just in time) compiler and let you debug the code yourself in LLDB!



> **Note**: This script will raise an error because the `--debug` option was turned on. If you were to use pdb to inspect the `res.GetError()`, you'll find that it contains the following message: *Execution was halted at the first instruction of the expression function because "debug" was requested...* This is OK and not part of an error you should worry about since you're debugging your own expression. It's worth noting that you'll not get a return value from this script since it errored out.

Now you can inspect, step, and even augment parameters just like you would any LLDB expression. Since you're in the Terminal window, you'll need to inspect the source code using the **`source list`**, or more conveniently, the **`list`** or **`l`** LLDB command.

In LLDB, type the following:

```
(lldb) l
```

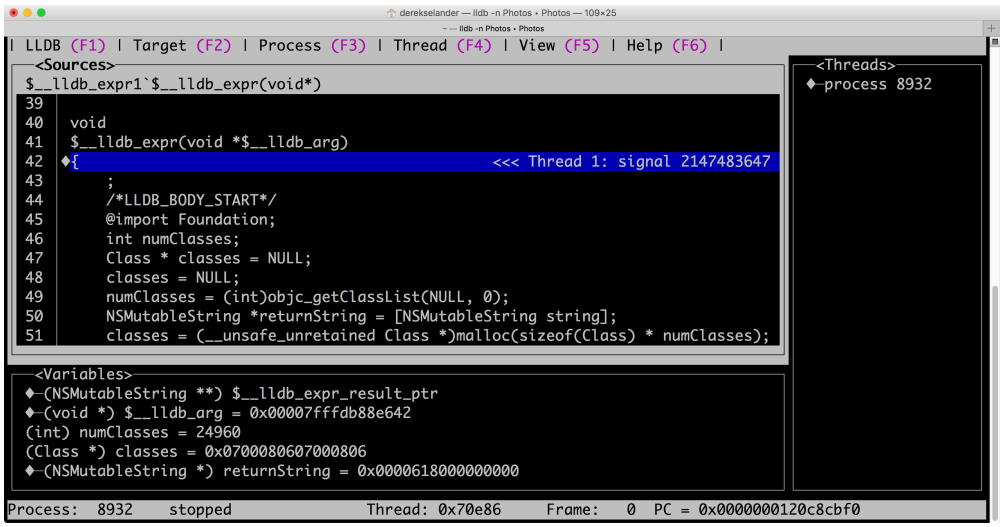
This will list the current line and slowly move down through the source file. Repeat to view the next set of lines.

```
(lldb) l
```

If you were to keep executing the same command, it would eventually cover all the source lines available and produce no more output. Another solution to viewing and stepping through source code while in a LLDB Terminal window is to use the **`gui`** LLDB command. This recently-added command in LLDB will transform your Terminal window into a `curses`-style GUI.

Type the following to jump into the LLDB GUI window:

```
(lldb) gui
```



From here, you can step through code using the **N** key, or step into code using **S**. Once you're at a location of interest, you can exit out of the LLDB GUI by typing `Fn + F1` (or just `F1` if you don't have the standard function keys enabled) to bring up the LLDB menu.

From there, press **X** to E**x**it out of the LLDB GUI and back into your console to print out/modify or alter control.

Using the `--debug` option is a great way to hunt for logic that returns unexpected results in your script that is running "actual" code — that is, JIT code — inside the process.

For example, if your script gave you unexpected results, I would get rid of all pdb instances, add the `-g` option to a `expression` command executed by `HandleCommand` and then execute the custom command I was working on. From there, I would use the LLDB console (through Terminal or through Xcode... which is a far better way to view the source code) and then hunt for the reason why my JIT code isn't returning the expected results.

> **Note**: It's worth noting I have occasionally experienced errors when using `po` LLDB command while exploring contents inside a paused JIT function created with the `-g` option. If that's the case, I'll fall back to using the **frame variable** command to explore the parameters of interest. Check out Chapter 6, "Thread, Frame & Stepping Around" to learn more about the `frame` LLDB command.

Once you're satisfied with exploring the `--debug` option, remove the `-g` option for your Python script.

How to handle problems

As I alluded to in the introduction to this chapter, you're going to run into problems when building these scripts. Let's recap what options you have, depending on the type of problem you encounter when building out these scripts.

Typically, you should perform iterative development on a Python script, save, then reload your script while LLDB is attached to a process and the process is still running.

Python build errors

When reloading your script, you might encounter something like this:



This is an example of a build error that occurred when I was creating my script. This command will not successfully load since there are Python syntax errors in it.

This is the most straightforward type of problem, because reloading the script will show me the error. I can tell that on line 37, I have unmatched indentation in the `findclass` Python script.

Python runtime errors or unexpected values

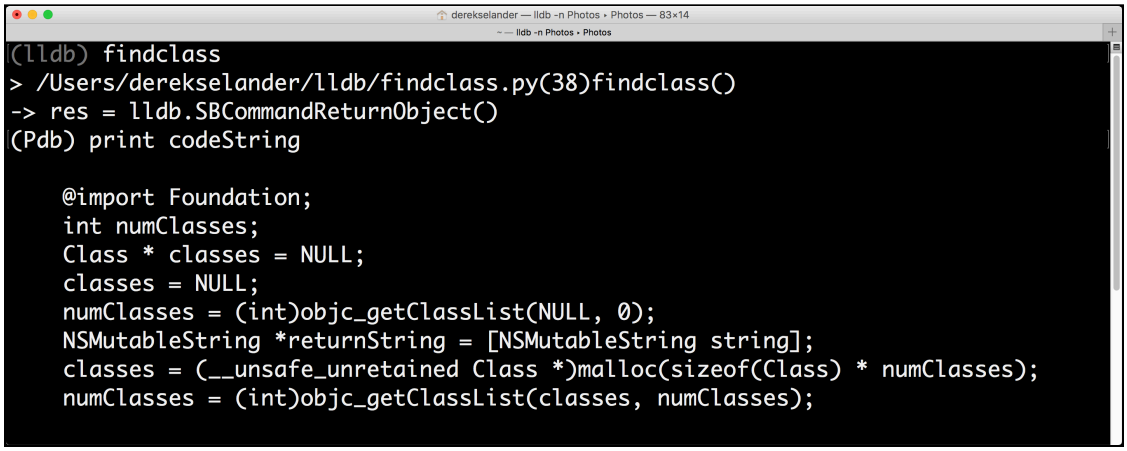
What if your Python script loads just fine, and you don't get any build errors to the console when reloading — but you receive unexpected output, or your script crashes and you need to further inspect what's happening?

Now, you can use the Python `pdb` module. Go to your Python script (in this case, **findclass.py**) and add the following line of code right before you expect the problem to occur:

```
import pdb; pdb.set_trace()
```

Jump over to Terminal (again, `pdb` will freeze Xcode, so Terminal is your only option for `pdb`) and attach to a process with LLDB, then try your command again.

From there, execution will eventually freeze and hit your `pdb`-triggered breakpoint, where you can inspect parameters and step through the flow of execution.



JIT code build errors

Often, you're executing actual code inside the process and then return the value back to your Python script. Again, this will be referred to as *JIT code* throughout the remainder of the book.

Imagine the following: you're executing a long batch of JIT code, and when running the JIT code in a `HandleCommand` method from the LLDB Python module you get an error saying something is not working.

This is one of the more annoying aspects with working with these scripts, since the debugger won't give you line information along with the error. If you can't uniquely identify where the error could have originated, you'll need to systematically comment out areas of your code until `HandleCommand` produces no errors for the JIT code.

From there, you can hone in on any locations giving you problems, and fix them.

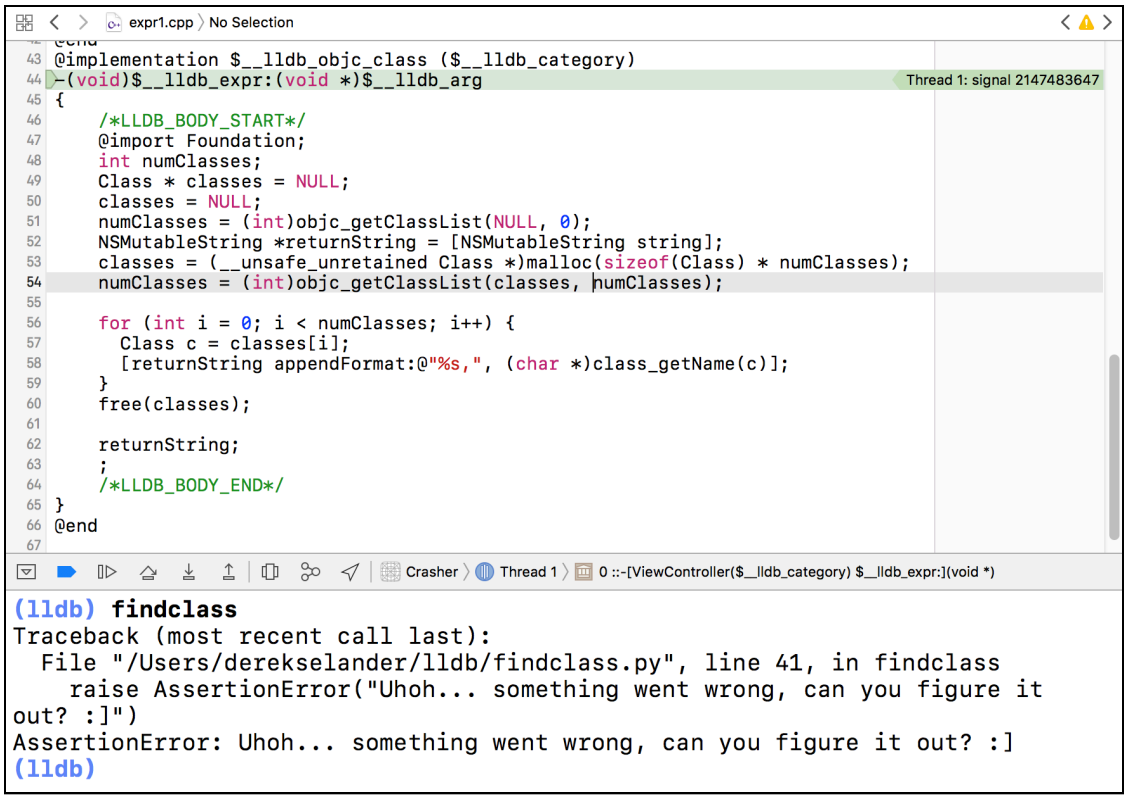
JIT code with unexpected results

The final types of errors you could encounter are unexpected results from your JIT code. For example, in the `findclass.py` script, what if you didn't get an expected class? What if you get more hits than you would have expected, searching for a particular query?

This is when that `--debug` option from the LLDB `expression` command comes in handy. Hunt down the method for `SBDebugger`'s or `SBCommandReturnObject`'s `HandleCommand` and add the `-g` option when the `expression` command is being used.

```
debugger.GetCommandInterpreter().HandleCommand("expression -lobjc -O -g
-- " + codeString, res)
```

Reload your script, then execute the command.



Control will stop on the JIT code and let you inspect it to determine what went wrong. If you do this in Xcode, you have all the conveniences of your hotkeys while viewing the source code to let you inspect and step over execution to hunt down the problem.

Where to go from here?

You're now equipped to tackle the toughest debugging problems while making your own custom scripts!

There's a lot more you can do with `pdb` than what I described here. Check out https://docs.python.org/2.7/library/pdb.html and read up on the other cool features of `pdb`. Be sure to remember that the version of `pdb` must match the version of Python that LLDB is using.

While you're at it, now's the time to start exploring other Python modules to see what other cool features they have. Not only do you have the lldb Python module, but you also have the full power of Python to use when creating advanced debugging scripts.

Chapter 23: Script Bridging Classes & Hierarchy

You've learned the essentials of working with LLDB's Python module, as well as how to correct any errors using Python's pdb debugging module.

In addition, you've explored `expression`'s `--debug` option to manually pause and explore JIT code that's being executed in-process. Now you'll explore the main players within the lldb Python module for a good overview of the essential classes.

You'll be building a more complex LLDB Python script as you learn about these classes. You'll create a regex breakpoint that only stops after the scope in which the breakpoint hit has finished executing. This is useful when exploring initialization and accessor-type methods, and you want to examine the object that's being returned after the function executes.

In this chapter, you'll learn how to create the functionality behind this script while learning about the major classes within the LLDB module. You'll continue on with this script in the next chapter by exploring how to add optional arguments to tweak the script based on your debugging needs.

The essential classes

Within the lldb module, there are several important classes:

- **`lldb.SBDebugger`**: The "bottleneck" class you'll use to access instances of other classes inside your custom debugging script.

  There will always be one reference to an instance of this class passed in as a function parameter to your script. This class is responsible for handling input commands into LLDB, and can control where and how it displays the output.

- **`lldb.SBTarget`**: Responsible for the executable being debugged in memory, the debug files, and the physical file for the executable resident on disk.

  In a typical debugging session, you'll use the instance of `SBDebugger` to get the selected `SBTarget`. From there, you'll be able to access the majority of other classes through `SBTarget`.

- **`lldb.SBProcess`**: Handles memory access (reading/writing) as well as the multiple threads within the process.

- **`lldb.SBThread`**: Manages the stack frames (`SBFrames`) within that particular thread, and also manages control logic for stepping.

- **`lldb.SBFrame`**: Manages local variables (given through debugging information) as well as any registers frozen at that particular frame.

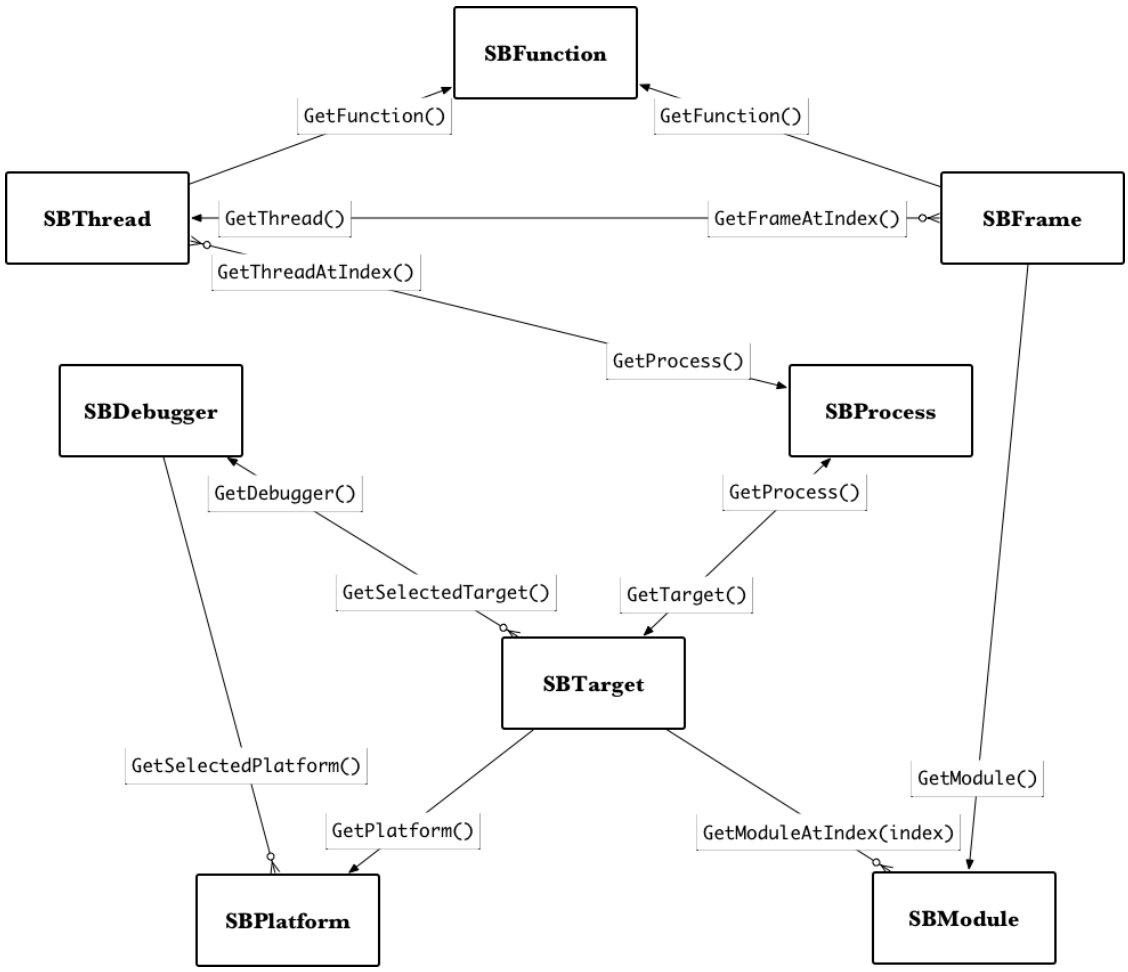
- **`lldb.SBModule`**: Represents a particular executable. You've learned about modules when exploring dynamic libraries; a module can include the main executable or any dynamically loaded code (like the Foundation framework).

  You can obtain a complete list of the modules loaded into your executable using the `image list` command.

- **`lldb.SBFunction`**: This represents a generic function — the code — that is loaded into memory. This class has a one-to-one relationship with the `SBFrame` class.

Got it? No? Don't worry about it! Once you see how these classes interact with each other, you'll have a better understanding of their place inside your program.

This diagram is a *simplified* version of how the major LLDB Python classes interact with each other.



If there's no direct path from one class to another, you can still get to a class by accessing other variables, not shown in the diagram, that point to an instance (or all instances) of a class (many of which are not shown in the diagram).

That being said, the entry-point into the majority of these objects will be through an instance of `SBDebugger`, passed in as an instance variable called `debugger` in your scripts. From there, you'll likely go after the `SBTarget` through `GetSelectedTarget()` to access all the other instances.

Exploring the lldb module through... LLDB

Since you'll be incrementally building a reasonably complex script over the next two chapters, you'll need a way to conveniently reload your LLDB script without having to stop, rerun and attach to a process. You'll create an alias for reloading the `~/.lldbinit` script while running LLDB.

Append the following to your `~/.lldbinit` file:

```
command alias reload_script command source ~/.lldbinit
```

This adds a command called `reload_script` which reloads the `~/.lldbinit` file. Now whenever you save your work, you can simply reload the updated contents without having to restart LLDB and the process it's attached to.

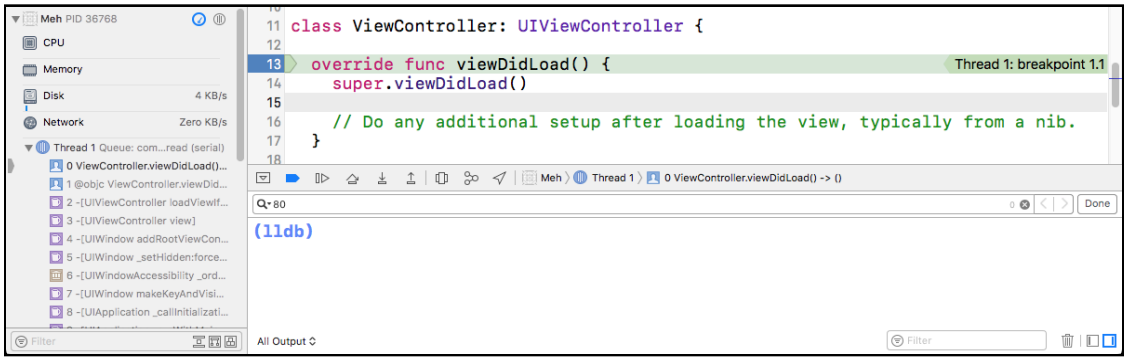
In addition, this is a useful command to ensure everything inside your `~/.lldbinit` file is still valid. Typically, errors in your `~/.lldbinit` will go unnoticed since LLDB doesn't have access to your stderr when it's starting up. However, reloading while LLDB is alive and active will dump any syntax errors in your scripts right to the LLDB console.

While you're building out this new script, you'll create a one-time-use burner project to explore these LLDB Python APIs. To mix things up, you'll create a **tvOS** project this time.

Open Xcode. Select **File\New\Project...** . Choose **tvOS\Single View Application**. Call this new project **Meh** (because I am out of creative names to use!). Make sure the language is set to **Swift**. Then save the project wherever you want.

Once the project has been created, open **ViewController.swift** and add a GUI breakpoint to the beginning of `viewDidLoad()`.

Build, run and wait for the breakpoint to be triggered. Jump over to the LLDB console.



Next, type the following into LLDB:

```
(lldb) script lldb.debugger
```

You'll get output similar to the following:

```
<lldb.SBDebugger; proxy of <Swig Object of type 'lldb::SBDebugger *' at
0x113f2f990> >
```

LLDB has a few easily accessible global variables that map to some of the classes described above:

- `lldb.SBDebugger` -> `lldb.debugger`
- `lldb.SBTarget` -> `lldb.target`
- `lldb.SBProcess` -> `lldb.process`
- `lldb.SBThread` -> `lldb.thread`
- `lldb.SBFrame` -> `lldb.frame`

You've just explored the global variable `lldb.debugger`. Now it's time to explore the other variables.

Type the following into LLDB:

```
(lldb) script lldb.target
```

You'll get output similar to the following:

```
<lldb.SBTarget; proxy of <Swig Object of type 'lldb::SBTarget *' at
0x1142daae0> >
```

This probably doesn't mean much to you at the moment because it's only displaying the instance of the class, and not the context of what it does, nor what it represents.

This is why the `print` command might be more useful when you're starting to explore these classes.

```
(lldb) script print lldb.target
```

This will give you some intelligible output to provide some context:

```
Meh
```

Using the `print` command is a useful trick when you want to get a summary of an instance, just as calling `po` on an object gives you an `NSObject`'s `description` method in Objective-C. If you didn't use the `print` command, you'd have to hone in on properties and attributes of `SBTarget` to figure out the name of the target.

> **Note**: It's fine that you're playing with global Python variables in one-line scripts. However, it's important you don't use these global variables in your actual Python scripts since you can modify the state (i.e step out of a function), and these global variables will not update until your script has finished.

> The correct way to reference these instances is to start from `SBDebugger`, which is passed into your script function, and drill down to the appropriate variable from there.

Go through the remainder of the major global variables and print them out. Start with the following:

```
(lldb) script print lldb.process
```

You'll get the following:

```
SBProcess: pid = 47294, state = stopped, threads = 7, executable = Meh
```

This printed out the process being run. As always, your data might differ (`pid`, `state`, `thread` etc...).

Next, type the following into LLDB:

```
(lldb) script print lldb.thread
```

This time you'll get something like this:

```
thread #1: tid = 0x13a921, 0x000000010fc69ab0
Meh`ViewController.viewDidLoad(self=0x00007fa8c5b015f0) -> () at
ViewController.swift:13, queue = 'com.apple.main-thread', stop reason =
breakpoint 1.1
```

This has printed out the thread that triggered the breakpoint.

Next, try the frame variable:

```
(lldb) script print lldb.frame
```

And finally, this one results in:

```
frame #0: 0x000000010fc69ab0
Meh`ViewController.viewDidLoad(self=0x00007fa8c5b015f0) -> () at
ViewController.swift:13
```

This will get you the specific frame where the debugger is paused. You could, of course, access other frames in other threads. These global variables are merely convenience getters for you. I would strongly recommend using these global LLDB variables when you're playing with and learning about these classes.

Check out http://lldb.llvm.org/python_reference/index.html to learn about which methods these classes implement.

Alternatively, you can use Python's `help` function to get the docstrings for a particular class. For example, if you were in the Xcode debugging console, and you wanted info on the active `SBTarget`, you could do this:

```
(lldb) script help(lldb.target)
```

Alternatively, you could go after the actual class instead of the global variable:

```
(lldb) script help(lldb.SBTarget)
```

Don't be afraid to ask for help from the `help` function. I use it all the time when I'm figuring out my plan of attack through the lldb module.

Learning & finding documentation on script bridging classes

Learning this stuff isn't easy. You're faced with the learning curve of the LLDB Python module, as well as learning Python along the way.

The best way to go about learning these foreign APIs is to start in easy, small steps. This means attaching to a process and using the **`script`** command to explore a class or API. Once you've mastered how to use a certain API, it's fair game to throw it into a custom Python script.

For example, if I stumbled across the `SBTarget` class and saw the global variable, `lldb.target`, I would jump to the URL https://lldb.llvm.org/python_reference/lldb.SBTarget-class.html and use the LLDB `script` command while exploring the online documentation.

```
(lldb) script lldb.target
<lldb.SBTarget; proxy of <Swig Object of type 'lldb::SBTarget *' at 0x111974270> >
(lldb) script lldb.target.GetTriple()
'x86_64h-apple-tvos'
(lldb) b -[UIViewController viewDidLoad]
Breakpoint 1: where = UIKitCore`-[UIViewController viewDidLoad], address = 0x000000010da609ef
(lldb) script lldb.target.ResolveLoadAddress(0x000000010da609ef)
<lldb.SBAddress; proxy of <Swig Object of type 'lldb::SBAddress *' at 0x11209dbd0> >
(lldb) script print lldb.target.ResolveLoadAddress(0x000000010da609ef)
UIKitCore`-[UIViewController viewDidLoad]
(lldb) script print lldb.target.GetNumBreakpoints()
2
(lldb) script print lldb.target.DeleteAllBreakpoints()
True
(lldb) script print lldb.target.GetNumBreakpoints()
0
(lldb) script lldb.target.module['UIKitCore']
<lldb.SBModule; proxy of <Swig Object of type 'lldb::SBModule *' at 0x111974270> >
(lldb) script print lldb.target.module['UIKitCore']
(x86_64) /Applications/Xcode.app/Contents/Developer/Platforms/AppleTVOS.platform/Developer/Library/CoreSimulator/
  Profiles/Runtimes/tvOS.simruntime/Contents/Resources/RuntimeRoot/System/Library/PrivateFrameworks/
  UIKitCore.framework/UIKitCore
(lldb)
```

Easy reading

I frequently find myself scouring the class documentation to see what the different classes can do for me with their APIs. However, doing that in the LLDB Terminal makes my eyes water. I typically jump to the online documentation because I am a sucker for basic Cascading Style Sheet(s) with more colors than just the background color and text color.

In fact, I do this *so* much, I often use this LLDB command to directly bring up any class I want to explore:

```
command regex gdocumentation 's/(.+)/script import os; os.system("open
https:" + unichr(47) + unichr(47) + "lldb.llvm.org" + unichr(47) +
"python_reference" + unichr(47) + "lldb.%1-class.html")/'
```

Stick this command in your **~/.lldbinit** file. Make sure the above command is only on one line or else this will not work.

This command is called **`gdocumentation`**; it takes a case-sensitive query and opens up the class of interest in your web browser. For example, if I installed this command into my **~/.lldbinit** file, and I was attached to a process and wanted to explore the online help documentation for `SBTarget`, I would type the following into LLDB:

```
(lldb) gdocumentation SBTarget
```

This will direct my web browser to the online documentation of `SBTarget`. Neat!

Documentation for the more serious

If you're one of those developers who really, really needs to master LLDB's Python module, or if you have plans to build a commercial product which interacts with LLDB, you'll need to take a more serious approach for digging through the lldb module APIs and documentation.

Since there's no search functionality available on http://lldb.llvm.org/python_reference/ (at the time of writing), you need a way to easily search all the classes for a particular query.

A drastic but excellent suggestion is to copy the entire http://lldb.llvm.org/python_reference/ site for offline storage using a tool like http://www.httrack.com/. From there, you can search using Terminal commands.

For example, if I scraped the entire site into `~/websites/lldb` on my computer and I wanted to search for all classes that had an API that pertained to `SBProcess`, I would type the following in Terminal:

```
mdfind SBProcess -onlyin ~/websites/lldb
```

It's not a bad idea to also go after the LLDB mailing lists found here http://lists.llvm.org/pipermail/lldb-dev/ and grab that website for offline use. There's are a *ton* of useful hints and explanations given by the authors of LLDB which are buried in the list's archives.

One final way to search for content is to use an often overlooked feature of Google to filter queries to a particular website using the **site:** keyword.

For example, if I wanted to search for all occurrences of `SBTarget` in LLDB's mailing archives, I could use the following query with Google:

```
SBTarget site:http://lists.llvm.org/pipermail/lldb-dev/
```

Fortunately, the next couple of chapters will guide you through most of the important classes, so the above suggestions are only meant for the crazy ones out there.

Creating the BreakAfterRegex command

It's time to create the command you were promised you'd build at the beginning of this chapter!

How would you design a command to stop immediately after a function, print out the return value, then continue? Take a bit of happy thinking time for yourself, and try to figure out how you'd go about creating this script.

I'm serious — stop reading until you've given this an honest attempt. I'll wait.

...

...

...

Good. What did you come up with?

When writing these types of scripts, it's always good practice to envision what you want to achieve, and work your way back from there.

You'll name your command script **BreakAfterRegex.py**. The steps the command needs to take are as follows:

- First, use LLDB to create a regex breakpoint.
- Next, add a breakpoint action to **step-out** of execution (from Chapter 6, "Thread, Frame & Stepping Around") until the current frame has finished executing.
- Finally, you'll use your knowledge of registers from Section II to print out the correct register that holds the return value.

Using your favorite text editor, create **BreakAfterRegex.py** in your **~/lldb** directory.

Once the file is created, open it and add the following:

```
import lldb

def __lldb_init_module(debugger, internal_dict):
  debugger.HandleCommand('command script add -f
BreakAfterRegex.breakAfterRegex bar')

def breakAfterRegex(debugger, command, result, internal_dict):
  print ("yay. basic script setup with input: {}".format(command))
```

You should know what this is doing by now — but in case you forgot, `__lldb_init_module` is a callback function called by LLDB after your script has finished loading into the Python address space.

From there, it references a `SBDebugger` instance passed in as `debugger` to execute the following line of code:

```
command script add -f BreakAfterRegex.breakAfterRegex bar
```

This will add a command named `bar` which is implemented by `breakAfterRegex` within the module `BreakAfterRegex` (named after the file, naturally). If you gave a silly command like `wootwoot` instead of `bar`, your LLDB command would be named that instead.

Open your `~/.lldbinit` file and append the following line:

```
command script import ~/lldb/BreakAfterRegex.py
```

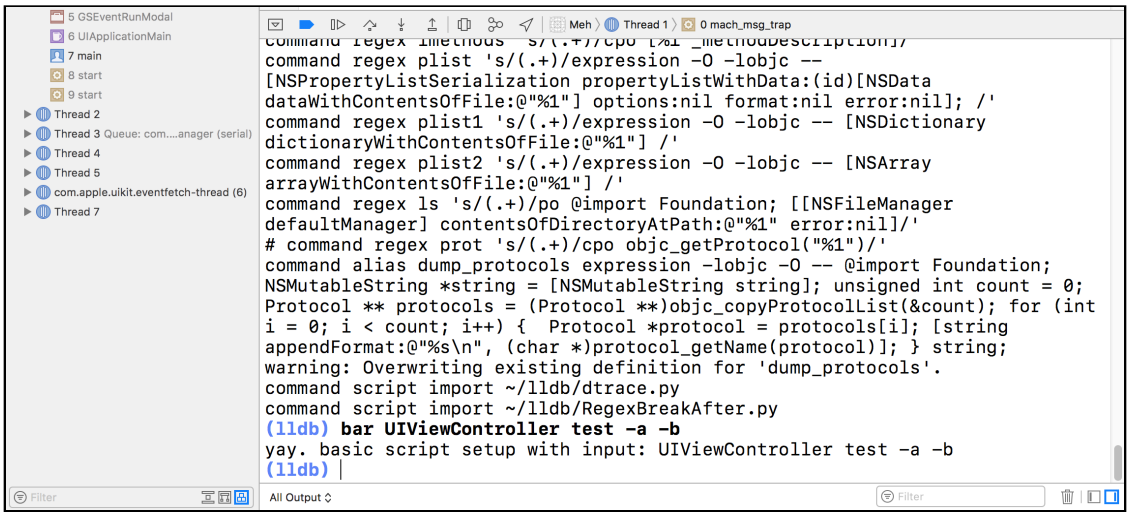
Save the file. Open Xcode, which should still be paused on `viewDidLoad()`. In the LLDB console, reload the script using your newly created convenience command:

```
(lldb) reload_script
```

You'll get a variable amount of output, as LLDB will display all the scripts it's loading. This will reload the contents in your `lldbinit` file and make the `bar` command functional.

Let's try out the `bar` command. In LLDB, type the following:

```
(lldb) bar UIViewController test -a -b
```



The output in your new LLDB script will echo back the parameters you've supplied to it.

You've got the basic skeleton up and working. It's time to write the code to create a breakpoint based upon your input. You'll start with creating input designed solely for handling the regular expression.

Head back to `BreakAfterRegex.py` and find `def breakAfterRegex(debugger, command, result, internal_dict):`.

Remove the `print` statement and replace it with the following logic:

```
def breakAfterRegex(debugger, command, result, internal_dict):
  # 1
  target = debugger.GetSelectedTarget()
  breakpoint = target.BreakpointCreateByRegex(command)

  # 2
  if not breakpoint.IsValid() or breakpoint.num_locations == 0:
    result.AppendWarning(
      "Breakpoint isn't valid or hasn't found any hits")
  else:
```

```
    result.AppendMessage("{}".format(breakpoint))

  # 3
  breakpoint.SetScriptCallbackFunction(
    "BreakAfterRegex.breakpointHandler")
```

Here's what you're doing:

1. Create a breakpoint using the regex input from the supplied parameter. The `breakpoint` object will be of type `SBBreakpoint`.
2. If breakpoint creation is unsuccessful, the script will warn you it couldn't find anything to break on. If successful, the breakpoint object is printed out.
3. Finally, the breakpoint is set up so the function `breakpointHandler` is called whenever the breakpoint hits.

What's that I hear you say? What's an `SBBreakpoint`? Well, you can look it up through LLDB!

```
(lldb) script help(lldb.SBBreakpoint)
```

If perusing the output in the LLDB console makes your eyes water, a more convenient way to view the documentation can be found here:

https://lldb.llvm.org/python_reference/lldb.SBBreakpoint-class.html.

If you installed the `gdocumentation` command mentioned earlier, you can simply type the following instead:

```
(lldb) gdocumentation SBBreakpoint
```

Grabbing the first line of the help documentation indicates an `SBBreakpoint` class represents a logical breakpoint and its associated settings.

OK — back on the main road after that little sightseeing trip. Where were we? Oh right — you haven't created the handler function that will be called when the breakpoint is hit. You'll do that now.

Right below `breakAfterRegex`, add the following function:

```
def breakpointHandler(frame, bp_loc, dict):
  function_name = frame.GetFunctionName()
  print("stopped in: {}".format(function_name))
  return True
```

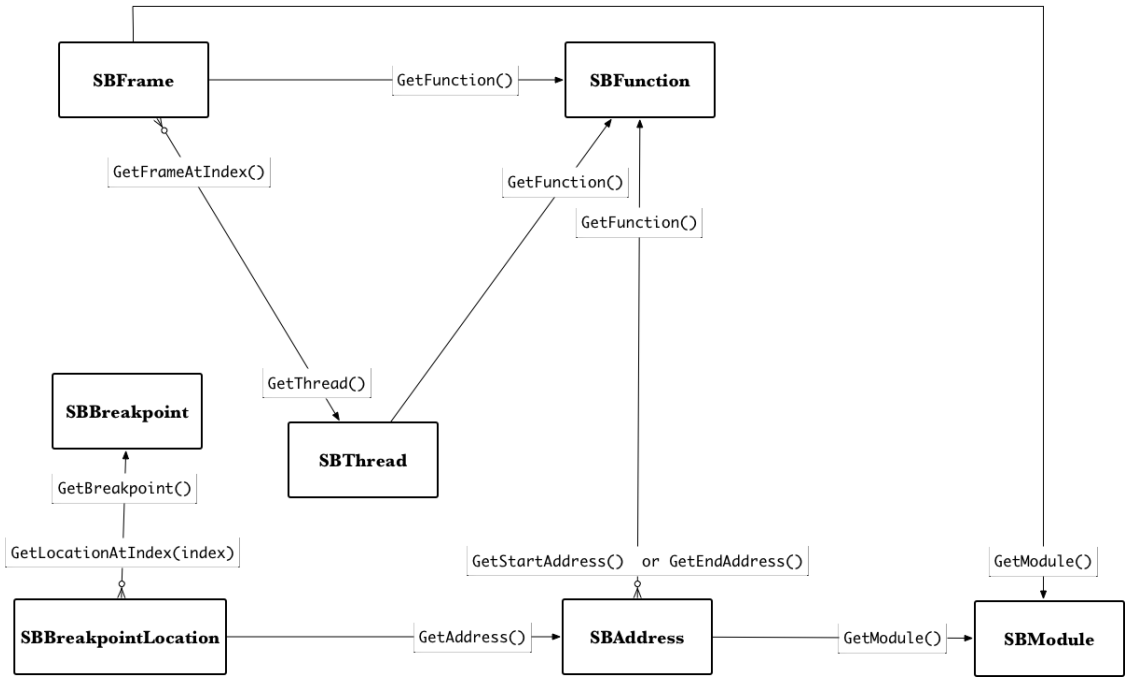
This function is called whenever any of the breakpoints you created using your new command are hit, and will then print out the function name. Notice the return of `True` at the end of the function. Returning `True` will result in your program stopping execution. Returning `False`, or even omitting a `return` statement will result in the program continuing to run after this method executes.

This is a subtle but important point. When creating callback functions for breakpoints (i.e. the `breakpointHandler` function you just created), you have a different method signature to implement. This consists of a `SBFrame`, `SBBreakpointLocation`, and a Python dictionary.

The `SBFrame` represents the frame you've stopped in. The `SBBreakpointLocation` is an instance of one of your breakpoints found in `SBBreakpoint`.

This makes sense because you could have many hits for a single breakpoint, especially if you try to break on a frequently implemented function, such as `main`, or if you use a well-matched regular expression.

Here's another diagram that showcases the simplified interaction of classes when you've stopped on a particular function:



As you (might have?) noticed, `SBFrame`, and `SBBreakpointLocation` are your lifelines to the majority of important lldb classes while in your breakpoint callback function. Using the previous diagram, you can get to all the major class instances through `SBFrame` or through `SBFrame`'s reference to `SBModule`.

Remember, you should never use `lldb.frame` or other global variables inside your scripts since they could hold a stale state while being executed in a script, so you must traverse the variables starting with the `frame`, or `bc_loc` to get to the instance of the class you want.

If you accidentally make a typo, or don't understand some code, simply insert a breakpoint in the script using the Python pdb module and work your way back from there. You learned about the pdb module in Chapter 22, "Debugging Script Bridging".

This script is starting to get complicated — looks like a good time to reload and test it out. Open the Xcode console window and reload your script:

```
(lldb) reload_script
```

Go through the motions of executing some commands again to test it out:

```
(lldb) bar somereallylongmethodthatapplehopefullydidntwritesomewhere
```

You'll get output similar to the following:

```
warning: Breakpoint isn't valid or hasn't found any hits
```

Ok, good. Time to try out an actual breakpoint. Let's go after a rather frequently executed method.

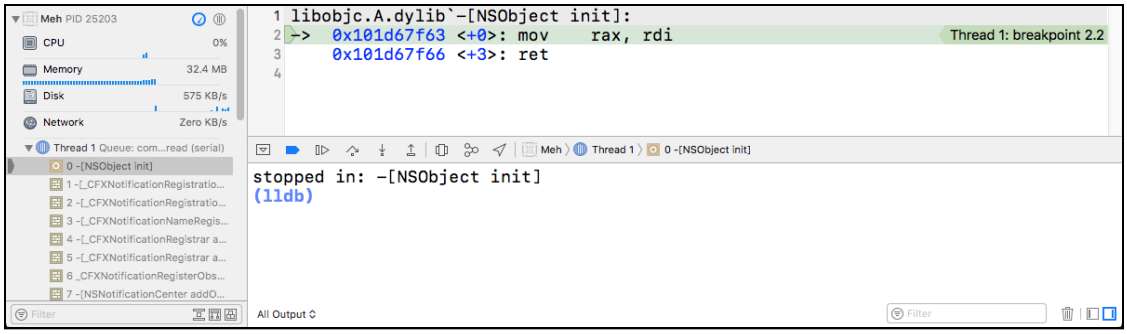
In the LLDB console type the following:

```
(lldb) bar NSObject.init\]
```

You'll see something similar to the following:

```
SBBreakpoint: id = 3, regex = 'NSObject.init\]', locations = 2
```

Continue execution and use the Simulator remote to click around the tvOS Simulator to trigger the breakpoint. If you're having trouble tripping the breakpoint, one surefire way is to navigate to the simulator's home screen. From the Simulator, **Hardware\Home** (or more easily, ⌘ + **Shift** + **H**).



Cool. You've successfully added a command to create a regex breakpoint! That's pretty darn neat-o.

Right now, you've stopped on one of `NSObject`'s `init` methods, which could be a class or an instance method. This is very likely a subclass of `NSObject`. You'll manually replicate the actions you're about to implement in the Python script using LLDB.

Using the LLDB console, finish executing this method:

```
(lldb) finish
```

Remember your register calling conventions? Since you're working on the tvOS Simulator and this architecture is x64, you'll want to use the **`RAX`** register. Print out the return value of `NSObject`'s `init` in LLDB.

```
(lldb) po $rax
```

Depending on where and how you were playing with the Simulator, you'll see a different object. I received the following output:

```
<_CFXNotificationNameWildcardObjectRegistration: 0x61000006e8c0>
```

If curiosity gets the better of you, feel free to explore the properties and methods within the class you just stumbled across using the strategies discussed in Chapter 17, "Exploring and Method Swizzling Objective-C Frameworks".

Stepping out and printing is the exact logic you'll implement now in your custom script callback function.

Open **BreakAfterRegex.py** and revisit the **`breakpointHandler`** function. Modify it to look like the following:

```
def breakpointHandler(frame, bp_loc, dict):
  # 1
  '''The function called when the regular
  expression breakpoint gets triggered
  '''

  # 2
  thread = frame.GetThread()
  process = thread.GetProcess()
  debugger = process.GetTarget().GetDebugger()

  # 3
  function_name = frame.GetFunctionName()

  # 4
  debugger.SetAsync(False)

  # 5
```

```
    thread.StepOut()

    # 6
    output = evaluateReturnedObject(debugger,
                                    thread,
                                    function_name)
    if output is not None:
      print(output)

    return False
```

B-B-B-B-B-Breakdown time!

1. Yep, if you're building a full-on Python command script, you've got to add some docstrings. You'll thank yourself later. Trust me.

2. You're climbing the hierarchical reference chain to grab the instance of `SBDebugger` and `SBThread`. Your starting point is through `SBFrame`.

3. This grabs the name of the parent function. Since you're about to step out of this current `SBFrame`, it's about to get invalidated, so grab any stack references you can before the stepping-out occurs.

4. `SetAsync` is an interesting function to use when tampering with control flow while scripting in a program. The debugger will run asynchronously while executing the program, so you need to tell it to synchronously wait until `stepOut` completes its execution before handing control back to the Python script.

   A good programmer will clean up the state to the async's previous value, but that becomes a little complicated, as you could run into threading issues when this callback function triggers if multiple breakpoints were to hit this callback function. This is not a noticeable setting change when you're debugging, so it's fine to leave it off.

5. You then step out of the method. After this line executes, you'll no longer be in the frame you previously stopped in.

6. You're calling a soon-to-be implemented method **`evaluateReturnedObject`** that takes the appropriate information and generates an output message. This message will contain the frame you've stopped in, the return object, and the frame the breakpoint stepped out to.

You're all done with that Python function! Now you need to implement `evaluateReturnedObject`. Add it below the previous function you just wrote:

```
def evaluateReturnedObject(debugger, thread, function_name):
  '''Grabs the reference from the return register
  and returns a string from the evaluated value.
  TODO ObjC only
  '''

  # 1
  res = lldb.SBCommandReturnObject()

  # 2
  interpreter = debugger.GetCommandInterpreter()
  target = debugger.GetSelectedTarget()
  frame = thread.GetSelectedFrame()
  parent_function_name = frame.GetFunctionName()

  # 3
  expression = 'expression -lobjc -O -- {}'.format(
      getRegisterString(target))

  # 4
  interpreter.HandleCommand(expression, res)

  # 5
  if res.HasResult():
    # 6
    output = '{}\nbreakpoint: '\
      '{}\nobject: {}\nstopped: {}'.format(
        '*' * 80,
        function_name,
        res.GetOutput().replace('\n', ''),
        parent_function_name)
    return output
  else:
    # 7
    return None
```

Here's what that does:

1. You first instantiate a new **`SBCommandReturnObject`**. You've seen this class already in your primary functions as the `result` parameter. However, you're creating your own here because you'll use this instance to evaluate and modify an expression. A typical `po "something"` will produce output, including two newlines, straight to the console. You need to grab this output before it goes to the console and remove those newlines… because you're fancy like that. In Chapter 25, "Script Bridging with SBValue & Language Contexts", you'll explore a cleaner alternative to evaluating code and obtaining output, but for now you'll make do with your existing knowledge of the `SBCommandReturnObject` class.

2. You grab a few variables for use later on.

3. Here you create the expression to be executed that prints out the return value. The **`getRegisterString`** is yet another unimplemented function you'll implement in just a moment — I promise this will be the last time I do that to you! This function will return the syntax needed to access the register which holds the return value.

   This is required because you can't know if this script is running on a watchOS, iOS, tvOS, or macOS device, so you'll need to augment the register name depending upon the architecture. Remember, you also need to use the Objective-C context, since Swift hides the registers from you!

4. Finally, you execute the expression through the debugger's command interpreter, **`SBCommandInterpreter`**. This class interprets your commands but allows you to control where the output goes, instead of immediately piping it to stderr or stdout.

5. Once `HandleCommand` has executed, the output of the expression should now reside in the `SBCommandReturnObject` instance. However, it's good practice to ensure the return object actually has any output to give to you.

6. If everything worked correctly, you format the old, stepped-out function along with the object and currently stopped function into a string and return that.

7. However, if there was no input to print from the `SBCommandReturnObject`, you return `None`.

One more method, and then you're (sort of) done! Implement `getRegisterString` at the bottom of your Python script:

```
def getRegisterString(target):
  triple_name = target.GetTriple()
  if "x86_64" in triple_name:
    return "$rax"
  elif "i386" in triple_name:
    return "$eax"
  elif "arm64" in triple_name:
    return "$x0"
  elif "arm" in triple_name:
    return "$r0"
  raise Exception('Unknown hardware. Womp womp')
```

You're using the `SBTarget` instance to call **`GetTriple`**, which returns a description of the hardware the executable is designed to run on. Next, you determine which syntax you need to access the register responsible for the return value based on your architecture. If it's an unknown architecture, then raise an exception.

You've done it! Save your work, jump back to Xcode and reload the script with your trusty `reload_script` command in the LLDB command line.

Next, before you get started with the full-blown command, remove all previous breakpoints like so:

```
(lldb) br del
About to delete all breakpoints, do you want to do that?: [Y/n] Y
All breakpoints removed. (1 breakpoint)
```

It's time to take this beauty for a spin!

Type the following into LLDB:

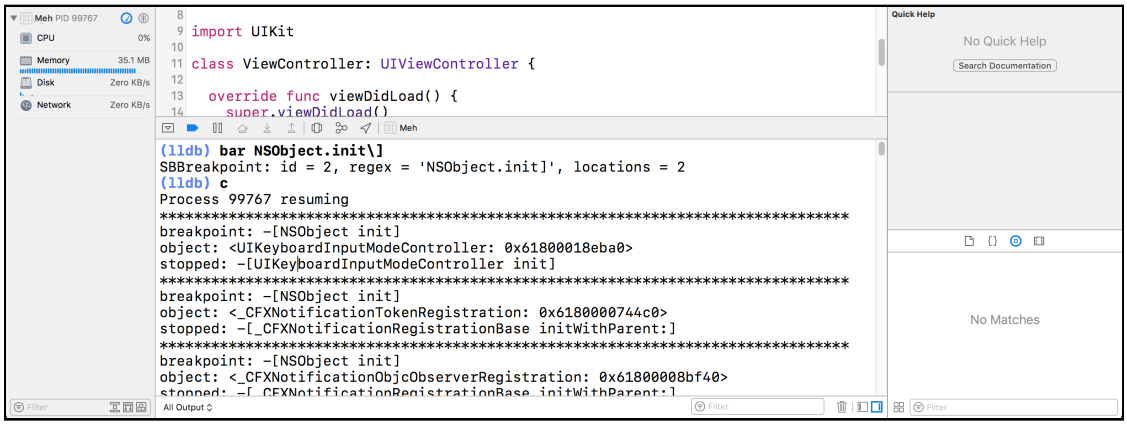
```
(lldb) bar NSObject.init\]
```

This time your script will execute your completed command's script when it hits the breakpoint.

Do whatever you need to do through the tvOS Simulator to trigger the `init` breakpoint; closing the application will work (⌘ + Shift + H), as will bringing up the Apple TV Remote (found in the **Hardware** menu) and tapping on the remote.

Once hit, you'll get some beautiful output which showcases the method you've stopped on (in this case `-[NSObject init]`), the object that is being created, and the calling method as well.

Since you've created a breakpoint on a frequently-called method, you'll soon hit the same breakpoint again.



This is a fun tool to have at your disposal. You could, for instance, create a well-crafted regex breakpoint to trigger each time an `NSURL` is created within any application… owned by you or not. For example, you could try:

```
(lldb) bar NSURL(\(\w+\))?\ init
```

The “weird” syntax is needed because a lot of the initialization methods for NSURL are in categories. Alternatively, you could use this script on a problematic getter method of a Core Data object that is returning unusual values.

Where to go from here?

You’ve begun your quest to create Python LLDB scripts of real-world complexity. In the next chapter, you’ll take this script even further and add some cool options to customize this script.

But for now, have fun and play around with this `bar` script! Attach LLDB to some applications running in the simulator and play around with the command. Try the already mentioned `NSURL` initialization (or `NSURLRequest` initialization) breakpoints.

Once you get bored of that, see what objects are using Core Data by inspecting the return value of `-[NSManagedObject valueForKey:]` or check out all the items that are being created from a nib or storyboard by breaking on an `initWithCoder:` method.

Chapter 24: Script Bridging with Options & Arguments

When you're creating a custom debugging command, you'll often want to slightly tweak functionality based upon options or arguments supplied to your command. A custom LLDB command that can do a job only one way is a boring one-trick pony.

In this chapter, you'll explore how to pass optional parameters (a.k.a. options) as well as arguments (parameters which are *expected*) to your custom command to alter functionality or logic in your custom LLDB scripts.

You'll continue working with the **`bar`** ("break-after-regex") command you created in the previous chapter. In this chapter, you'll finish up the `bar` command by adding logic to handle options in your script.

By the end of this chapter, the `bar` command will have logic to handle the following optional parameters:

- **Non-regular expression search**: Using the **`-n`** or **`--non_regex`** option will result in the `bar` command using a non-regular expression breakpoint search instead. This option will *not* take any additional parameters.
- **Filter by module**: Using the **`-m`** or **`--module`** option will only search for breakpoints in that particular module. This option will expect an additional parameter which specifies the name of the module.
- **Stop on condition**: Using the **`-c`** or **`--condition`** option, the `bar` command will evaluate the given condition after stepping out of the current function. If `True`, execution will stop. If `False`, execution will continue. This option will expect an additional parameter which is a string of code that will be executed and evaluated as an Objective-C `BOOL`.

This will be a dense but fun chapter. Make sure you've got a good supply of caffeine!

Setting up

If you've gone through the previous chapter and your `bar` command is working, then you can continue using that script and ignore this part. Otherwise, head on over to the **starter** folder in this chapter's resources, and copy the **BreakAfterRegex.py** file into your **~/lldb** folder. Make sure your **~/.lldbinit** file has the following line which you should have from the previous chapter:

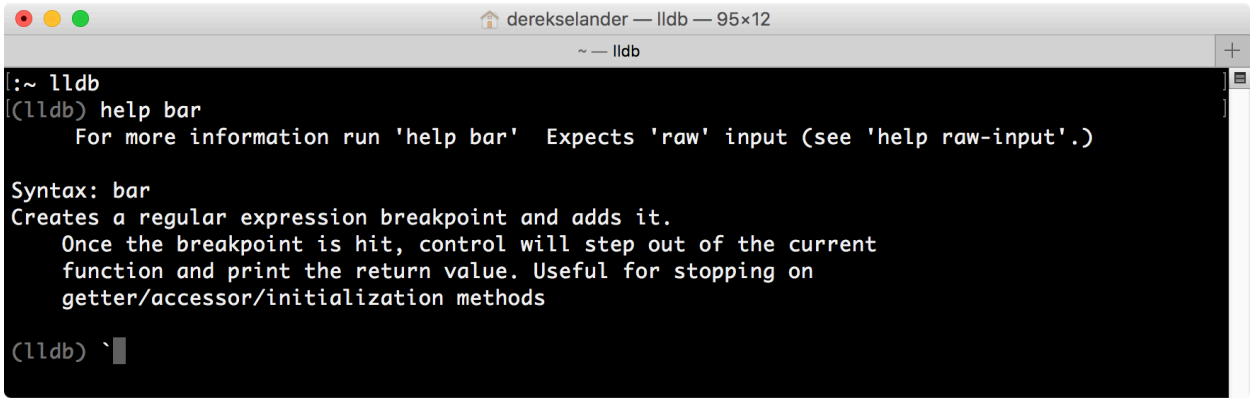
```
command script import ~/lldb/BreakAfterRegex.py
```

If you've any doubts if this command loaded successfully into LLDB, simply fire up a new LLDB instance in Terminal:

```
lldb
```

Then check for the help docstring of the `bar` command:

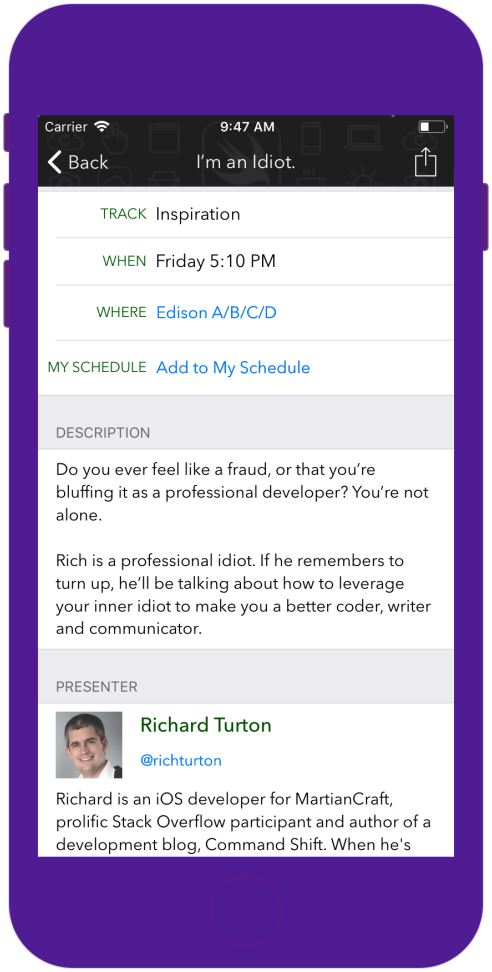
```
(lldb) help bar
```



If you get an error, it's not successfully loaded; but if you got the docstring, you're golden.

The RWDevCon project

For this chapter, you'll use an app called **RWDevcon**. It's a live app, available in the App Store (https://itunes.apple.com/us/app/rwdevcon-the-tutorial-conference/id958625272).



This app is the companion app for the **RWDevcon** conference, https://www.rwdevcon.com/, where it's an annual tradition to see how many times you can touch Ray Wenderlich's shoulders before he gets annoyed. Try it! My personal best is 37!

For this project, I've forked from commit `84167c68` which can be found in the **starter** folder. However, you can get a more up-to-date version here: https://github.com/raywenderlich/RWDevCon-App.

Navigate to the starter folder then open, build, then run this application. Take a look around to get acquainted with the project.

There's no need to explore any of the source code. With the aid of the `bar` command, you'll be able to explore different items of interest with smart breakpoint queries.

But before we can do that, let's talk about how to make this bar command much more powerful.

The optparse Python module

The lovely thing about LLDB Python scripts is you have all the power of Python — and its modules — at your disposal.

There are three notable modules that ship with Python 2.7 that are worth looking into when parsing options and arguments: **`getopt`**, **`optparse`**, and **`argparse`**.

`getopt` is kind of low level and `optparse` is on its way out since it's been deprecated after Python 2.7. Unfortunately `argparse` is mostly designed to work with Python's `sys.argv` — which is not available to your Python LLDB command scripts. This means `optparse` will be your go-to option. Facebook's Chisel, Apple's own custom LLDB scripts, and I all use this module. So, it's kinda the de-facto standard for parsing arguments. ;]

The `optparse` module will let you define an instance of type **OptionParser**, a class responsible for parsing all your arguments. For this class to work, you need to declare what arguments and options your command supports. This makes sense because optional parameters may or may not take additional values for that particular option.

Take a brief look at an example. Consider the following:

```
some_command woot -b 34 -a "hello world"
```

The command is named `some_command`. But what are the arguments and options being passed into this command?

If you didn't give any context to the parser, then this statement is ambiguous. The parser doesn't know whether or not the `-b` or `-a` option should take in parameters for the option. For example, the parser could think this command is passed three arguments: `['woot', '34', 'hello world']`, and two options `-b`, `-a` with no parameters. However, if the parser expected `-b` and `-a` to take parameters, the parser would give you the argument of `['woot'], '34'` for the `-b` option and `'hello world'` for `-a`.

Let's dive into `optparse` some more, and see how we can use it to handle cases like this.

Adding options without params

With the knowledge you need to educate your parser with what arguments are expected, it's time to add your first option which will alter the functionality of the `bar` command to apply the `SBBreakpoint` without using a regular expression, but instead use a normal expression.

This argument will be backed by a Python `boolean` value, so no parameters are needed for this option. The existence (or lack thereof) of this option is all the information you need to determine the `boolean` value. If the argument exists, then it'll be `True`. Otherwise, `False`.

It's worth noting some script authors will engineer an option that will encourage a `boolean` option which explicitly requires a parameter for the Boolean value and default to either `True` or `False` if the option is not supplied.

For example, the following command takes an option, **`-f`** with no parameters:

```
some_command -f
```

This would then turn into:

```
some_command -f1
```

That's not really my style. But you might want to consider this design decision if you're building scripts for a wider audience, since it gives the user more explicit intentions.

Ok, enough chit-chat. Let's get to implementing this parser thing.

Open up **BreakAfterRegex.py** and add the following `import` statements at the top of the file:

```
import optparse
import shlex
```

The `optparse` is the module you just covered that contains the `OptionParser` class to parse any extra input given to your command.

The **`shlex`** module has a nice little Python function that conveniently splits up the arguments supplied to your command on your behalf while keeping string arguments intact.

For example, consider the following Python code:

```
import shlex
command = '"hello world" "2nd parameter" 34'
shlex.split(command)
```

This will produce the following output:

```
['hello world', '2nd parameter', '34']
```

This returns a Python `list` of parsed Python `strs`.

But before you go using this `split` method, you'll need to create the parser itself. Head to the very bottom of **BreakAfterRegex.py** and create the following method:

```
def generateOptionParser():
  '''Gets the return register as a string for lldb
    based upon the hardware
  '''
  usage = "usage: %prog [options] breakpoint_query\n" +\
          "Use 'bar -h' for option desc"
  # 1
  parser = optparse.OptionParser(usage=usage, prog='bar')
  # 2
  parser.add_option("-n", "--non_regex",
                    # 3
                    action="store_true",
                    # 4
                    default=False,
                    # 5
                    dest="non_regex",
                    # 6
                    help="Use a non-regex breakpoint instead")
  # 7
  return parser
```

Let's break this down, parameter by parameter:

1. You're creating the `OptionParser` instance and supplying it a **usage** param and a **prog** param. The `usage` will get displayed if you screw up and give the parser an argument it doesn't know how to handle. The `prog` option is used to address the name of the program. I always incorporate it because it resolves a weird little issue which lets you run the `-h` or `--help` option to get all the supported options for a custom command. If the `prog` arg is not in there, the `-h` command will not work correctly. It's one of life's little mysteries. `¯\_(ツ)_/¯`

2. This line (followed by the next four lines of non-commented code) add the **`--non_regex`** or **`-n`** parameter to the `parser`.

3. The **`action`** param informs what action should be done when this param is supplied. `"store_true"` informs the parser to store the Python Boolean `True` when this option is supplied.

4. The **`default`** param informs that the initial value will be `False`. If this option is not given, this will be the value.

5. The **`dest`** parameter will determine the name, **non_regex**, that you're giving to the property when the `OptionParser` parses your input. For example, consider the following code which parses a Python string of options and arguments in `command`:

```
command_args = shlex.split(command)
(options, args) = parser.parse_args(command_args)
options.non_regex
```

As you'll see shortly, the `parse_args` method produces a Python tuple containing a list of options (called `options`) and a list of arguments (called `args`). The `options` variable will now contain the `non_regex` property.

6. **`help`** will give you help documentation. You can get all the parameters and their info with the `--help` option. For example, when this is correctly set up in the `bar` command, all you have to do is type `bar -h` to see a list of all the options and what they do.

7. Once you've created the `OptionParser` and added the `-n` option, you're returning the instance of the `OptionParser`.

You've just created a method that will generate this `OptionParser` instance you need to start parsing those arguments. Now it's time to use this thing.

Jump back to the beginning of the `breakAfterRegex` function. Remove the following two lines:

```
target = debugger.GetSelectedTarget()
breakpoint = target.BreakpointCreateByRegex(command)
```

Then, in their place, add the following code:

```
'''Creates a regular expression breakpoint and adds it.
Once the breakpoint is hit, control will step out of the
current function and print the return value. Useful for
stopping on getter/accessor/initialization methods
'''

# 1
command = command.replace('\\', '\\\\')
# 2
command_args = shlex.split(command, posix=False)
```

```
  # 3
  parser = generateOptionParser()

  # 4
  try:
    # 5
    (options, args) = parser.parse_args(command_args)
  except:
    result.SetError(parser.usage)
    return

  target = debugger.GetSelectedTarget()

  # 6
  clean_command = shlex.split(args[0])[0]

  # 7
  if options.non_regex:
    breakpoint = target.BreakpointCreateByName(
                         clean_command)
  else:
    breakpoint = target.BreakpointCreateByRegex(
                         clean_command)

  # The rest remains unchanged
```

Make sure you have your indentation correct! This should be indented by two spaces, or whatever your single-tab width of choice is, as it's all part of the function.

Here's what that code does:

1. When parsing your input to the `OptionParser`, it will interpret slashes as escaping characters. For example, `"\'"` is interpreted as just `"'"`. This means you'll need to escape any backslash characters in your commands.

2. As you learned in a previous chapter, the `command` parameter passed into your custom LLDB scripts is a Python `str`, which contains all input that is passed into your argument. You'll pass this variable into the `shlex.split` method to obtain a Python `list` of Python `str`s. In addition, there's that `posix=False` which helps combat any input which contains special characters like a dash; otherwise, `OptionParser` will incorrectly assume that's an option being passed in. This is important because Objective-C has dashes in instance methods, so you don't want the dash to be incorrectly interpreted as an option!

3. Using the newly created `generateOptionParser` function, you create a parser to handle the command's input.

4. Parsing input can be error-prone. Python's usual approach to error handling is throwing exceptions. It's no surprise that `optparse` throws if it finds an error. If you don't catch exceptions in your scripts, LLDB will go down, which will also tank the process! Therefore, the parsing is contained in a try-except block to prevent LLDB from dying due to bad input.

5. The `OptionParser` class has a **`parse_args`** method. You're passing in your `command_args` variable to this method, and will receive a tuple in return. This tuple consists of two values: **`options`**, which consists of all option arguments (i.e. only the `non_regex` option right now). The other half of the tuple hands you all of the **`args`** which consists of any other input parsed by the parser.

6. You're taking the first captured argument (the breakpoint query) and assigning it to a variable called **`clean_command`**. Remember that `posix=False` mentioned in bullet 2? That logic will maintain the quotes around your captured argument which preserves your exact syntax. If you didn't have that `posix=False`, you could just use `args[0]`, but then you'd forfeit a lot of power in your regex by not being able to use the escape backslash character in your regex query.

7. You're putting your first option to use! You're checking the truthiness of `options.non_regex`. If `True`, you'll execute the `BreakpointCreateByName` method in `SBTarget` to implement a non-regular expression breakpoint. If the `non_regex` is `False` (by default it is when you supplied the **default** parameter inside the `generateOptionParser` function), then your script will use a regex search. Again, all you need to do is add the `–n` to your input for the `bar` command to make the `non_regex` `True`.

Testing out your first option

Enough code. Time to test this script out.

Instead of using that **reload_script** command you've used in the previous chapters, you'll try an alternative tactic that you might appreciate to reload the script.

Jump to Xcode and create a new symbolic breakpoint.

Make sure the **Breakpoint Navigator** tab is selected, then hunt down that lonely + icon in the lower left corner. Then select **Symbolic breakpoint...**. Alternatively for you cool kids, **`⌘ + Ctrl + \`**

In the **Symbol** section put **getenv**.

Add **two actions**. The first action adds the following command:

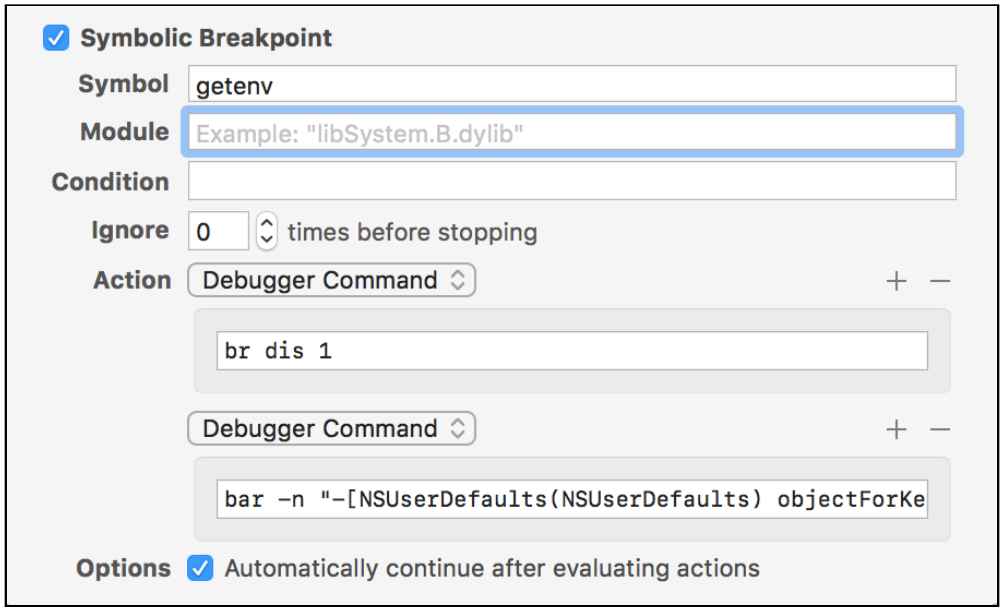
```
br dis 1
```

In the next action, add your `bar` command:

```
bar -n "-[NSUserDefaults(NSUserDefaults) objectForKey:]"
```

Finally select **Automatically continue after evaluating actions**.

When all is said and done, your symbolic breakpoint should look like this:



Can you figure out what you've just done? You've created a Symbolic breakpoint on the `getenv` C function. If I want to setup breakpoints before "my" code starts executing, or before reverse engineering an app, this is a good go-to to hook any logic for custom commands you want in LLDB.

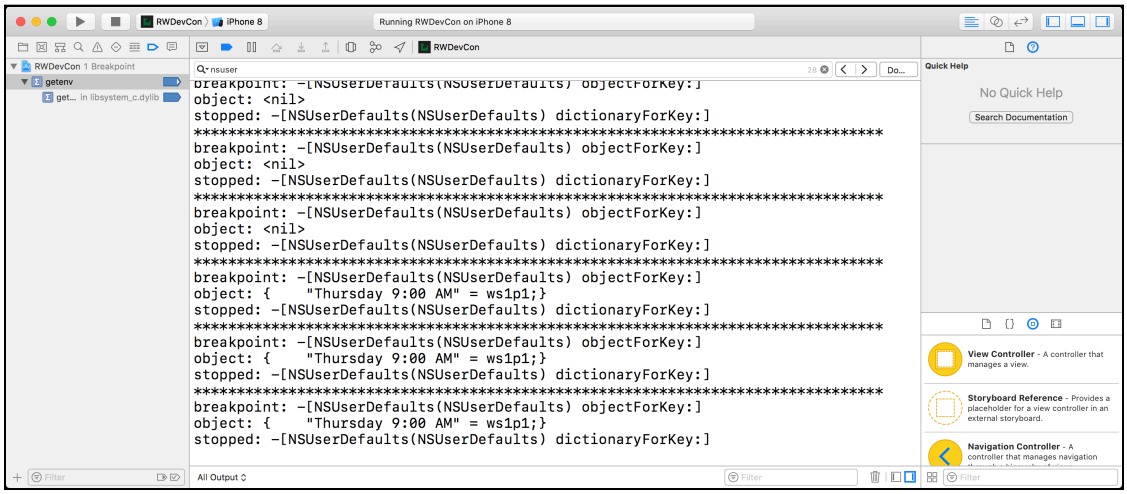
I'm not a fan of using `main`, since a lot of executables contain the function `main`, and the primary executable's `main` symbol might be stripped in a production build of an executable. We know that `getenv` will get hit for sure and will get hit before my code starts running.

What about those actions? The first action says to get rid of that `getenv` breakpoint. You're not deleting it; you're just disabling it. This is ideal since `getenv` gets called a fair bit and you need to get rid of this breakpoint once you've setup your LLDB logic. The use of **1** is mentioned because this breakpoint is the first breakpoint created for this session, which disables this symbolic breakpoint after it has run once.

After that, you're creating a non regular expression breakpoint on `NSUSerDefaults`'s `objectForKey:` method. We expect this method to return an `id` or `nil`, so let's see what this RWDevCon app is reading (or writing) to our `NSUserDefaults`.

Build and run the application.

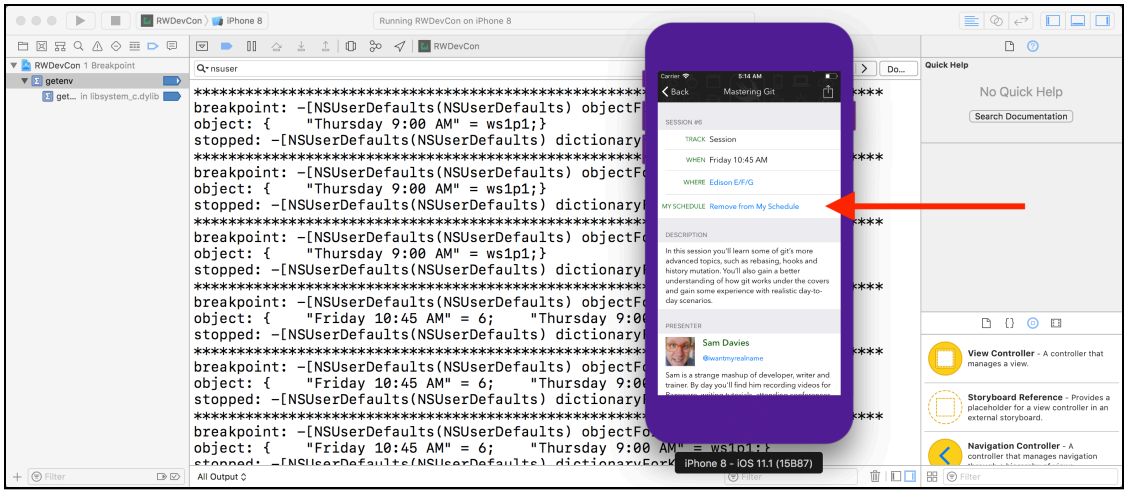
If you haven't taken a deep dive into the app, you'll likely get a lot of `nil` values. This means that this method is definitely getting read by some code in this app.



Tap on any one of the workshops to bring up the detail view controller.

Before you continue, clear the LLDB window (⌘ + **K**).

From there, tap **Add to my Schedule** while keeping an eye on the console output.



You can see there's an object that gets added to the `NSUserDefaults` that matches the `When` time.

Adding options with params

You've learned how to add an option that expects no arguments. You'll now add another option that expects a parameter. This next option will be the **`--module`** option to specify which module you want to constrain your regular expression query to.

This is very similar to `breakpoint set`'s `-s` (aka `--shlib` option) option where it expects the name of the module immediately after the option. You explored this back in Chapter 4, "Stopping in Code."

In the **BreakAfterRegex.py** script jump back down to the `generateOptionParser` function and add the following code right before `return parser`:

```
# 1
parser.add_option("-m", "--module",
                  # 2
                  action="store",
                  # 3
                  default=None,
                  # 4
                  dest="module",
                  help="Filter a breakpoint by only searching within a
specified Module")
```

1. You're adding a new option `-m` or `--module` to the OptionParser instance.
2. In the previous option, the `action` was `"store_true"`; this time it is `"store"`. This means this option expects a parameter.
3. This parameter's default value is `None`.
4. The name of this property will be **`module`**.

Jump back to the **`breakAfterRegex`** function and scan for the following lines:

```
if options.non_regex:
  breakpoint = target.BreakpointCreateByName(clean_command)
else:
  breakpoint = target.BreakpointCreateByRegex(clean_command)
```

Add **`options.module`** as the second parameter to both of these functions.

```
if options.non_regex:
  breakpoint = target.BreakpointCreateByName(clean_command,
options.module)
else:
  breakpoint = target.BreakpointCreateByRegex(clean_command,
options.module)
```

So how does this work? Let's print out the method signature right now for **BreakpointCreateByRegex**. Type the following in LLDB:

```
(lldb) script help (lldb.SBTarget.BreakpointCreateByRegex)
```

This will dump the small amount of documentation for this function. Although there is no help documentation for this method, it does give you a list of its method signatures.

```
(lldb) script help (lldb.SBTarget.BreakpointCreateByRegex)
Help on method BreakpointCreateByRegex in module lldb:

BreakpointCreateByRegex(self, *args) unbound lldb.SBTarget method
    BreakpointCreateByRegex(self, str symbol_name_regex, str module_name = None) ->
SBBreakpoint
    BreakpointCreateByRegex(self, str symbol_name_regex) -> SBBreakpoint
    BreakpointCreateByRegex(self, str symbol_name_regex, LanguageType symbol_language,
        SBFileSpecList module_list, SBFileSpecList comp_unit_list) -> SBBreakpoint

(lldb)
```

The following signature is worth discussing:

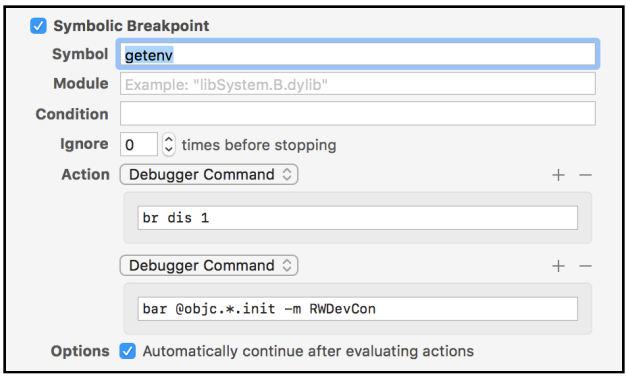
```
BreakpointCreateByRegex(SBTarget self, str symbol_name_regex, str
module_name=None) -> SBBreakpoint
```

Take note of the final parameter: `module_name=None`. The fact it's an optional parameter means if you don't supply a parameter, the `module_name` will take the value as `None`. This means when the `OptionParser` instance parses the options, you can supply `options.module` into the `BreakpointCreateByRegex` method regardless, since the default value of `options.module` will be `None`, which is the same as not applying an extra argument.

Time to test this out. Save your work in your script. Jump over to Xcode and modify that `getenv` Symbolic breakpoint. Replace the second action with the following line of code:

```
bar @objc.*.init -m RWDevCon
```

Make sure that `'C'` in `'Con'` is capitalized!



This will create a regex breakpoint on all Objective-C objects that are subclassed by a Swift object and stick a breakpoint on their initializer. You are filtering this breakpoint query to only search for breakpoints inside the **RWDevCon** module.

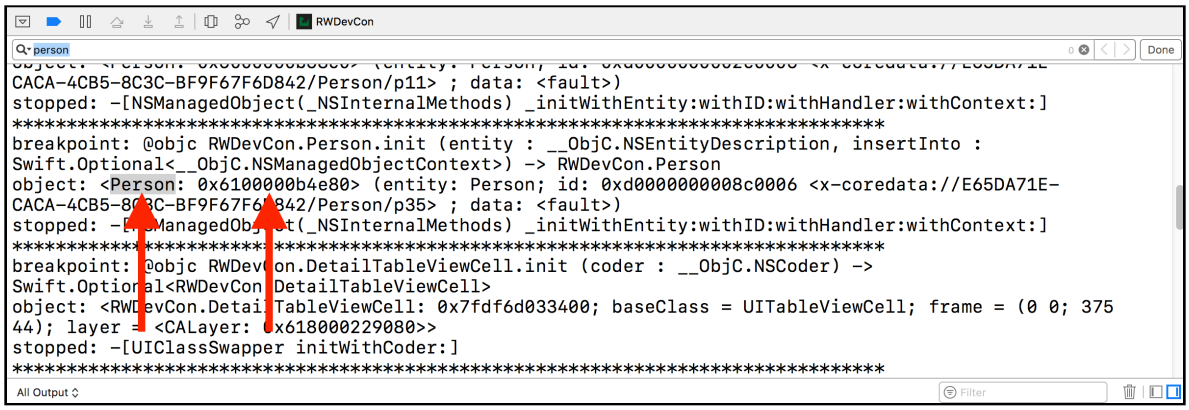
Run the application and check out all the Objective-C objects that are subclassed by Swift objects.

Take a quick look at the output. You'll get a lot of `__ObjC.NSEntityDescription` hits. That must mean there's some CoreData logic that's written in Swift, right?

Right!

Clear the screen (you should know that shortcut by now) and tap on a table cell that contains a workshop (i.e. no lunch or party dates) and see what pops up on the detail view controller.

You'll get a list of all the Objective-C objects that are subclassed by Swift. Search for the class named **`Person`**.



Copy the address into your clipboard.

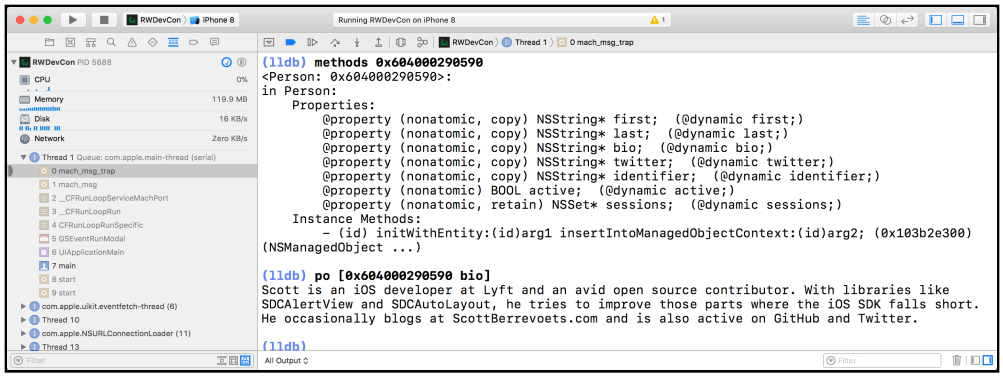
Before you paste in your address, let's dump all the methods implemented by this `Person` class. Since it's an Objective-C subclass, it's fair game to all those introspection commands you've made earlier.

In LLDB type the following:

```
(lldb) methods Person
```

This will dump all the methods the **Person** class implements that the Objective-C runtime knows about. Note that I said *Objective-C runtime*. There still could be Swift methods that this class implements that the Objective-C runtime doesn't know about *even* if the class inerhits from `NSObject`!

You can of course execute any of these methods on this valid `Person` instance.



Let's up the ante. You'll now create an option in the `bar` command that will allow you to add a condition, evaluated after the function the breakpoint is in finishes executing. If true, execution will stop; if false, execution will keep on going.

You'll apply this condition to `fullName` and only stop when you hit the name "Ray Wenderlich". Sneaky!

Passing parameters into the breakpoint callback function

Time to create the parser option for `-c`, or `--condition`!

Jump back to `BreakAfterRegex.py` and find `generateOptionParser`. Add the following line of code right before the `return parser` line of code:

```
parser.add_option("-c", "--condition",
                  action="store",
                  default=None,
                  dest="condition",
                  help="Only stop if the expression matches True. Can
reference return value through 'obj'. Obj-C only.")
```

You should know what this is doing now, but here's a quick recap. You're creating the `--condition` option which defaults to `None` and expects a parameter. The help text has something interesting in there. You're indicating you can reference the return value through the variable name **`obj`**. This means when you're evaluating code, you'll take the return register and assign `obj` to it.

Time to use this new option. But hold on... Think about this for a second. How are you going to pass the option parameters into the `SBBreakpoint` callback function?

Remember, this callback function is being called by a "private" C++ API and is limited to a specific method signature. Consider the following declaration where you set the breakpoint handler:

```
breakpoint.SetScriptCallbackFunction("BreakAfterRegex.breakpointHandler")
```

When the `SBBreakpoint` callback hits, this function will get called:

```
def breakpointHandler(frame, bp_loc, dict):
  # method contents here
```

You only have the `SBFrame`, `SBBreakpointLocation`, and an internal Python `dict` to work with to pass around information. How can this function read the parameters which are parsed by your `OptionParser` instance and be given into another function? This function signature is locked-in to only supply these parameters.

Several ideas come to mind to get around this problem. You can search for alternatives in `SBBreakpoint` or similar classes to see if there's an API that lets you pass in other params.

Alternatively, you can try and subclass a `SBBreakpoint` to add additional functionality to pass around the condition option parameter, or you can try using a global variable to pass around the parsed options. If you're really desperate, you can try and dynamically creating a method at runtime using the **exec** Python function.

Unfortunately, `SBBreakpoint` has no APIs to handle working with classes and callbacks, global variables are a bad idea in general and you could also run into threading problems for stale logic if multiple breakpoint callbacks are referencing a global set of options.

Subclassing won't work, since this Python LLDB class is dynamically generated behind the scenes by C++ code, and you'll get a new instance each time when trying to access the passed around `SBBreakpoint`. Besides, 99% of the time, using `exec` is just a bad, bad idea.

What's a developer to do?

This means you'll have to default to using global variables and deal with the global variable state. Consider the following situation. You assign the options to a global variable and create `SBBreakpoint` 1. You do the exact same thing for `SBBreakpoint` 2.

However, `SBBreakpoint` 1 gets triggered and the callback function is called, which references the global options. Since `SBBreakpoint` 2 was created, it has since modified these options to the incorrect expectation.

Fortunately, there's a slightly better alternative to using global variables, and you'll come up with a sneaky solution to resolve the global state of the options.

Instead of a global variable, you'll create a Python class, which will have a class property to hold the options being passed around.

Now to address that global state: instead of a property to hold the options, you'll use a Python **dict** to hold the options.

The nice thing about breakpoints is regardless of how many you create or delete, each breakpoint will have a unique ID per run session. This means you can use the breakpoint's ID as a unique key to reference a particular set of options for each breakpoint.

You can then set the breakpoints ID as the key and the options for that breakpoint as the value. Cool, right?

Jump to the top of **BreakAfterRegex.py** and add the following logic right underneath the import statements:

```
# 1
class BarOptions(object):

  # 2
  optdict = {}

  # 3
  @staticmethod
  def addOptions(options, breakpoint):
    key = str(breakpoint.GetID())
    BarOptions.optdict[key] = options
```

Going over this step-by-step:

1. You're declaring a class named `BarOptions` which inherits from type `object`. Think of `object` as Python's equivalent for `NSObject`. This class provides base functionality and generally makes your life a little easier. It's absolutely possible to not have a base class (just like in Swift), but some Python APIs play a little nicer when inheriting from `object`.

2. You're declaring a class variable named **`optdict`**. If you were to declare an instance variable, it would have to be inside an init function. Since you're only working with this class variable, you won't be setting up any initialization methods for this class.

3. You're also declaring a class method called **addOptions** (think `+[` in Objective-C or `class func` in Swift), which uniquely assigns the options that are bound to the `SBBreakpoint`'s ID.

Jump down to `breakAfterRegex` and add the following line of code right before the point where you specify the callback function (i.e. the call to `SetScriptCallbackFunction`):

```
BarOptions.addOptions(options, breakpoint)
```

After you've added this new line of code, create a new function to evaluate the condition. Add the new function **`evaluateCondition`** to the bottom of **BreakAfterRegex.py**:

```
def evaluateCondition(debugger, condition):
  '''Returns True or False based upon the supplied condition.
  You can reference the NSObject through "obj"'''

  # 1
  res = lldb.SBCommandReturnObject()
  interpreter = debugger.GetCommandInterpreter()
  target = debugger.GetSelectedTarget()

  # 2
  expression = 'expression -lobjc -O -- id obj = ((id){}); ((BOOL)
{})'.format(getRegisterString(target), condition)
  interpreter.HandleCommand(expression, res)

  # 3
  if res.GetError():
    print(condition)
    print('*' * 80 + '\n' + res.GetError() + '\ncondition:' + condition)
    return False
  elif res.HasResult():
    # 4
    retval = res.GetOutput()

    # 5
    if 'YES' in retval:
      return True

  # 6
  return False
```

Breaking that down:

1. You're creating a `SBCommandReturnObject` to handle the code being passed in from the `condition` parameter.
2. This will create and execute the custom expression that's being passed in. Notice you're declaring the instance variable **`obj`** and casting it to type **`id`** from the return register. This lets you conveniently reference the return value as `obj` instead of a hardware-specific register. The expression you provide will be cast into an Objective-C `BOOL`, which will either return a `YES` or `NO` output.
3. You'll evaluate the return value, and if it contains an error, print the error out. You're explicitly returning `False` or `True` within this function because you'll use this

return value to determine if execution should stop or not when evaluating this expression. Remember, the `SBBreakpoint` callback function `breakpointHandler` will stop execution if the function returns `True`. Execution will not stop if not `True` (i.e. `False`, `None` or no return) is returned.

4. This will assign the output to a variable named **`retval`** if there is one to grab.

5. It really pains me to teach expression parsing this way, since there's a much cleaner method of evaluating objects using `SBValues`, which you'll learn about in the next chapter. For now, you'll continue using the `SBCommandReturnObject` and compare the output to what you expect. If the expression is evaluated to `YES`, then pause execution.

6. If the execution returns `NO`, then just keep on executing by returning `False`.

Final round of code! Find **`breakpointHandler`** function. Add the following code beneath the `thread.StepOut()` call:

```
  # 1
  key = str(bp_loc.GetBreakpoint().GetID())
  # 2
  options = BarOptions.optdict[key]
  # 3
  if options.condition:
    # 4
    condition = shlex.split(options.condition)[0]
    # 5
    return evaluateCondition(debugger, condition)
```

Last explanation. Yay!

1. The `bp_loc` is of type `SBBreakpointLocation`. This class lets you reference the initial `SBBreakpoint` by the **`GetBreakpoint`** method. From there, you can reference the ID, which will be a number. Therefore, you need to cast this number as a Python `str` and assign that to the variable **`key`**.

2. This will grab the options from the class property **`optdict`** and assign it to the variable **`options`**.

3. Check if the `options` variable contains a non-`None` reference. If there's a valid reference, execute the logic.

4. This will unwrap the `condition` passed into the command line option. Again, you have to do a little extra work thanks to that `posix=False` mentioned earlier, but it allows you to use backslash and dash characters in our options & arguments.

5. Finally, you're calling the function **`evaluateCondition`** you created in the previous code snippet. You are returning the function's return value which will influence if execution should stop or not.

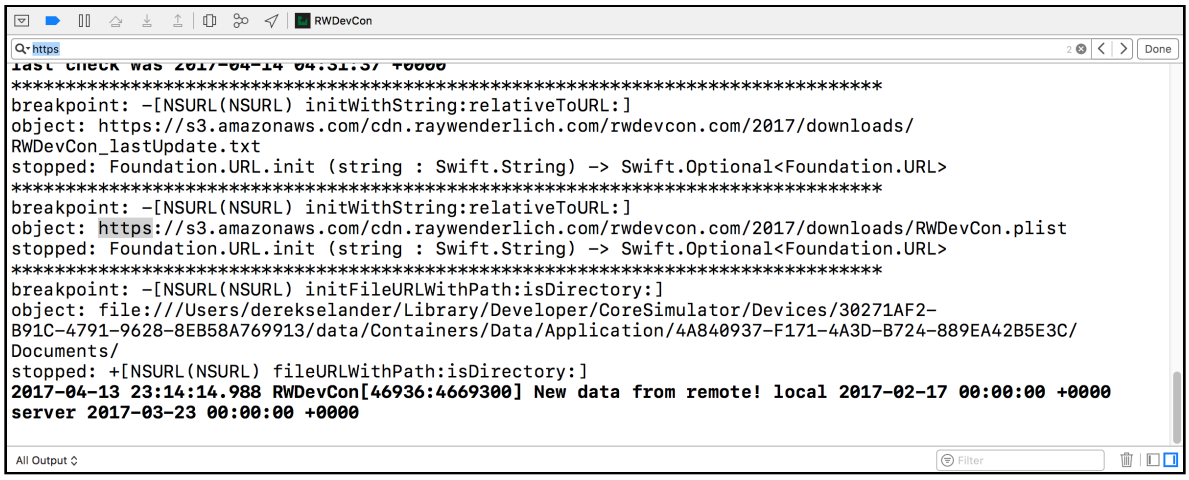
No more Python code (well, for this chapter...muwahahaha)! Save your work and head back to Xcode.

Again, modify the second action in the `getenv` symbolic breakpoint. This time, change it to the following:

```
bar NSURL\(.*init
```

This will breakpoint will now fire on the initialization of `NSURL`s. That weird syntax is necessary because the majority of `NSURL` initialization methods are created through categories.

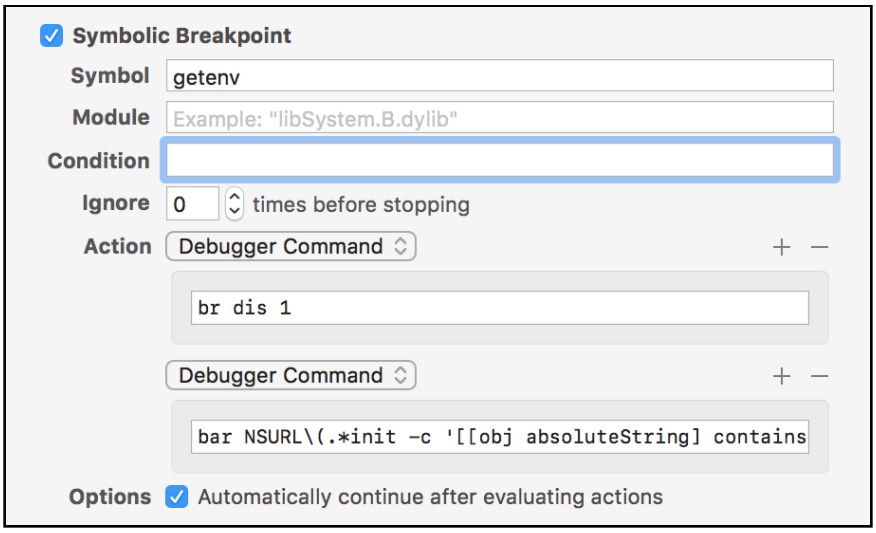
Scan for any HTTPS `NSURL`s in the console output.



Looks like the app is hitting some Amazon S3 webservice. Use the newly created `--condition` option of the `bar` command you've just created to stop when an `NSURL` returns from initialization and contains `"amazon"` in the `absoluteString`.

Go back after the `getenv` symbolic breakpoint and change the second action yet again to the following:

```
bar NSURL\(.*init -c '(BOOL)[[obj absoluteString]
containsString:@"amazon"]'
```



Build and run and see what happens…

Execution will stop on the exact line containing this `NSURL` … er… `URL`, since it stopped in the Swift context. But let's be real, that instance is a `NSURL`.

There are many creative situations the `bar` command can be utilized. Try to come up with some on your own!

Where to go from here?

That was pretty intense, but you've learned how to incorporate options into your own Python scripts.

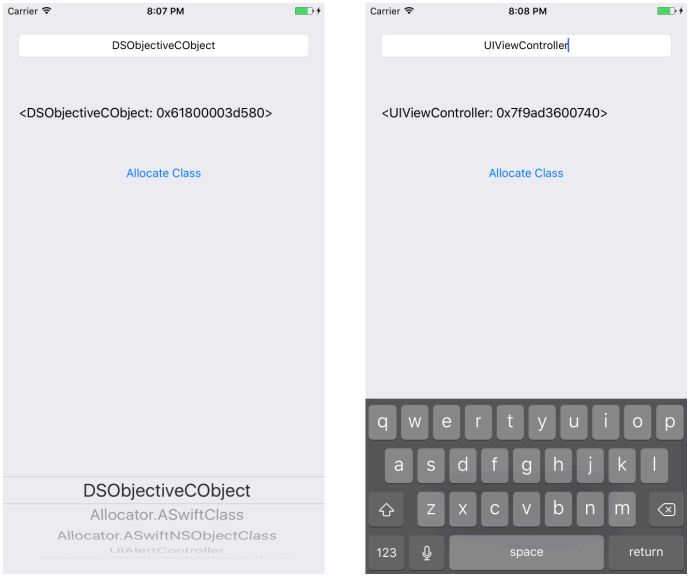
In the very unlikely chance you still have energy after reading this chapter, you should implement some sort of backtrace option for the `bar` command. There are many times, when debugging, where I wish I'd known the stack trace of an interesting object!

Chapter 25: Script Bridging with SBValue & Memory

So far, when evaluating JIT code (i.e. Objective-C, Swift, C, etc. code that's executed through your Python script), you've used a small set of APIs to evaluate the code.

For example, you've used `SBDebugger` and `SBCommandReturnObject`'s **`HandleCommand`** method to evaluate code. `SBDebugger`'s `HandleCommand` goes straight to `stderr`, while you have a little more control over where the `SBCommandReturnObject` result ends up. Once evaluated, you had to manually parse the return output for anything of interest. This manual searching of the output from the JIT code is a bit unsightly. Nobody likes stringly typed things!

So, it's time to talk about a new class in the lldb Python module, **`SBValue`**, and how it can simplify the parsing of JIT code output. Open up the Xcode project named **Allocator** in the **starter** folder for this chapter. This is a simple application which dynamically generates classes based upon input from a text field.



This is accomplished by taking the string from the text field and using it as an input to the `NSClassFromString` function. If a valid class is returned, it's initialized using the plain old `init` method. Otherwise, an error is spat out.

Build and run the application on any iOS 12 Simulator. You'll make zero modifications to this project, yet you'll explore object layouts in memory through `SBValue`, as well as manually with pointers through LLDB.

A detour down memory layout lane

To truly appreciate the power of the `SBValue` class, you're going to explore the memory layout of three unique objects within the Allocator application. You'll start with an Objective-C class, then explore a Swift class with no superclass, then finally explore a Swift class that inherits from `NSObject`.

All three of these classes have three properties with the following order:

- A `UIColor` called **`eyeColor`**.
- A language specific string (`String`/`NSString`) called **`firstName`**.
- A language specific string (`String`/`NSString`) called **`lastName`**.

Each instance of these classes is initialized with the same values. They are:

- `eyeColor` will be `UIColor.brown` or `[UIColor brownColor]` depending on language.
- `firstName` will be `"Derek"` or `@"Derek"` depending on language.
- `lastName` will be `"Selander"` or `@"Selander"` depending on language.

Objective-C memory layout

You'll explore the Objective-C class first, as it's the foundation for how these objects are laid out in memory. Jump over to the **DSObjectiveCObject.h** and take a look at it. Here it is for your reference:

```
@interface DSObjectiveCObject : NSObject

@property (nonatomic, strong) UIColor *eyeColor;
@property (nonatomic, strong) NSString *firstName;
@property (nonatomic, strong) NSString *lastName;

@end
```

As mentioned earlier, there are three properties: `eyeColor`, `firstName`, and `lastName` in that order.

Jump over to the implementation file **DSObjectiveCObject.m** and give it a gander to understand what's happening when this Objective-C object is initialized:

```
@implementation DSObjectiveCObject

- (instancetype)init
{
  self = [super init];
  if (self) {
    self.eyeColor = [UIColor brownColor];
    self.firstName = @"Derek";
    self.lastName = @"Selander";
  }
  return self;
}
@end
```

Nothing too crazy. The properties will be initialized to the values just described above.

When this is compiled, this Objective-C class will actually look like a C struct. The compiler will create a struct similar to the following pseudocode:

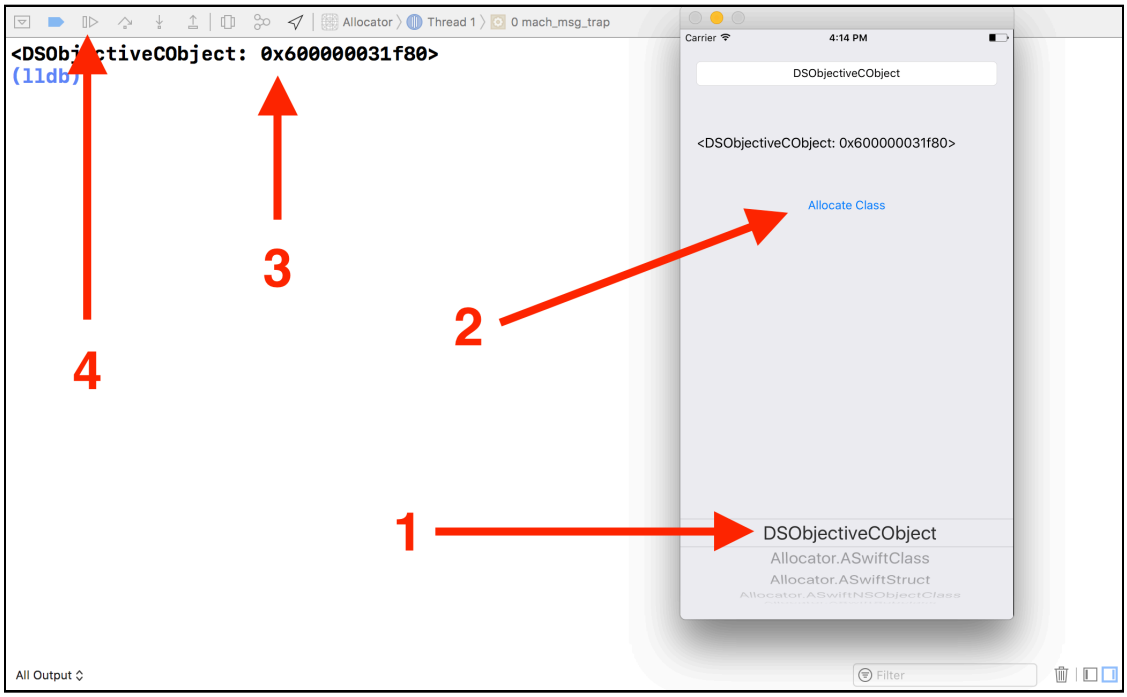
```
struct DSObjectiveCObject {
  Class isa;
  UIColor *eyeColor;
  NSString *firstName
  NSString *lastName
}
```

Take note of the `Class isa` variable as the first parameter. This is the magic behind an Objective-C class being considered an Objective-C class. This `isa` value is always the first value in an object instance's memory layout, and is a pointer to the class the object is an instance of. After that, the properties are added to this struct in the order they were written in your source code.

Let's see this in action through LLDB. Perform the following steps:

1. Make sure the **DSObjectiveCObject** is selected in the `UIPickerView`.
2. Tap on the **Allocate Class** button.
3. Once the reference address is spat out in the console, copy that address to your clipboard.

4. Pause execution and bring up the LLDB console window.



An instance of the `DSObjectiveCObject` has been created. You'll now use LLDB to spelunk into offsets of this object's contents.

Copy the memory address from the console output and make sure `po`'ing it will give you a valid reference (e.g. you're not stopped on a Swift stack frame when printing out this address).

For my case, I got the pointer `0x600000031f80`. As always, yours will be different. Print out the address through LLDB:

```
(lldb) po 0x600000031f80
```

You should get this expected line of output:

```
<DSObjectiveCObject: 0x600000031f80>
```

Since this can be treated as a C struct, you'll start spelunking into offsets of this pointer's contents.

In the LLDB console, type the following (replacing the pointer with yours):

```
(lldb) po *(id *)(0x600000031f80)
```

This casts the memory address to a pointer-to-an-`id` and then dereferences it. This will therefore give you access the object's **`isa`** pointer.

You should see this output:

```
DSObjectiveCObject
```

That's the class object's description, as expected.

Let's look at another way of viewing this memory. Use the **x** command (aka `examine`, a port from GDBs popularity with this command) to jump to the starting pointer, then `po` it. Enter the following:

```
(lldb) x/gx 0x600000031f80
```

This command says the following:

- Examine the memory (`x`)
- Print out the size of a **giant** word, (64 bits, or 8 bytes) (`g`)
- Finally, format it in hexadecimal (`x`).

If, hypothetically, you only wanted to view the first byte at this location in binary instead, you could type `x/bt 0x600000031f80` instead. This would be interpreted as examine (`x`), a byte (`b`) in binary (`t`). The `examine` command is definitely one of those nice commands to keep in your toolkit when exploring memory.

You'll see the following output (or at least, similar output, as the values will be different for you):

```
0x600000031f80: 0x0000000108b06568
```

This gives you output that tells you the value at memory address `0x600000031f80` contains `0x0000000108b06568`. Well, it does for me!

Jumping back to the task at hand, take the address printed out by the `x/gx` command and print out this new address using `po`.

```
(lldb) po 0x0000000108b06568
```

Once again, this will print out the `isa` class, which is the `DSObjectiveCObject` class. This is an alternative way to print out the `isa` instance, which might give more insight into what's happening. However, that took two LLDB commands instead of one, so you'll stick to dereferencing the pointer and not use the `x/gx` command.

Let's jump a little further into the `eyeColor` property. In the LLDB console:

```
(lldb) po *(id *)(0x600000031f80 + 0x8)
```

This says "start at `0x600000031f80` (or equivalent), go up 8 bytes and get the contents pointed at by this pointer." You'll get the following output:

```
UIExtendedSRGBColorSpace 0.6 0.4 0.2 1
```

How did I get to the number 8? Try this out in LLDB:

```
(lldb) po sizeof(Class)
```

The `isa` variable is of type `Class`. So by knowing how big a `Class` is, you know how much space that takes up in the struct, and therefore you know the offset of `eyeColor`.

> **Note**: When working with 64-bit architectures (x64 or ARM64), all pointers will be 8 bytes. In addition, the `Class` class itself is a pointer to a C struct not defined in the headers. This means in 64-bit architecture, all you need to do to move between different pointers is to jump by 8 bytes!
>
> There are types which are different sizes in bytes, such as `int`, `short`, `bool` and others, and the compiler may pad that memory to fit into a predefined size. However, there's no need to worry about that for now, since this `DSObjectiveCObject` class only contains pointers to `NSObject` subclasses, along with the `Class` object held in the `isa` variable.

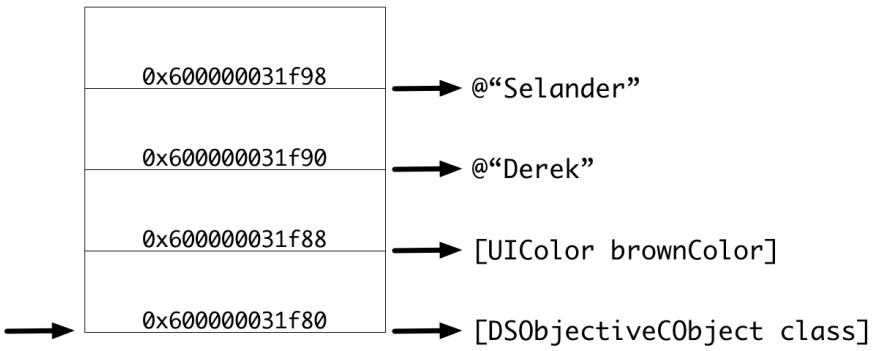
Keep on going. Increment the offset by another 8 bytes in LLDB:

```
(lldb) po *(id *)(0x600000031f80 + 0x10)
```

You're adding another 8 to get 0x10 in hexadecimal (or 16 in decimal). You'll get `@"Derek"`, which is the contents of the `firstName` property. Increment by yet another 8 bytes to get the `lastName` property:

```
(lldb) po *(id *)(0x600000031f80 + 0x18)
```

You'll get `@"Selander"`. Cool, right? Let's visually revisit what you just did to hammer this home:



You started at a base address that pointed to the instance of `DSObjectiveCObject`. For this particular example, this starting address is at `0x600000031f80`. You started by dereferencing this pointer, which gave you the `isa` variable, then you jumped by offsets of 8 bytes to the next Objective-C property, dereferenced the pointer at that offset, cast it to type `id` and spat it out to the console.

Spelunking memory is a fun and instructional way to see what's happening behind the scenes. This lets you appreciate the `SBValue` class even more. But you're not at the point of talking about the `SBValue` class, as you still have two more classes to explore. The first is a Swift class with no superclass, and the second is a Swift class which inherits from `NSObject`. You'll explore the non superclass Swift object first.

Swift memory layout with no superclass

> **Note**: It's worth mentioning right up front: the Swift ABI is still fluctuating. This means the information below could change before the Swift ABI completes. The day a new version of Xcode breaks the information in the following section, feel free to complain in ALL CAPS in the forums!

Time to explore a Swift class with no superclass! In the Allocator project, jump to **ASwiftClass.swift** and take a look at what's there.

```
class ASwiftClass {
  let eyeColor = UIColor.brown
  let firstName = "Derek"
  let lastName = "Selander"

  required init() { }
}
```

Here, you have the Swift equivalent for `DSObjectiveCObject` with the obvious "Swifty" changes.

Again, you can imagine this Swift class as a C struct with some interesting differences from its Objective-C counterpart. Check out the following pseudocode:

```
struct ASwiftClass {
  Class isa;

  // Simplified, see "InlineRefCounts"
  // in https://github.com/apple/swift
  uintptr_t refCounts;

  UIColor *eyeColor;

  // Simplified, see "_StringGuts"
```

```
  // in https://github.com/apple/swift
  struct _StringCore {
    uintptr_t _object;    // packed bits for string type
    uintptr_t rawBits;    // raw data
  } firstName;

  struct _StringCore {
    uintptr_t _object;    // packed bits for string type
    uintptr_t rawBits;    // raw data
  } lastName;
}
```

Pretty interesting right? You still have that `isa` variable as the first parameter.

After the `isa` variable, there's an eight byte variable reserved for reference counting and alignment called **`refCounts`**. This differs to your typical Objective-C object which doesn't contain this varible at this offset.

Next, you have the normal `UIColor`, but that's where this `ASwiftClass` struct goes completely off the rails.

A Swift String is a very interesting "object". In fact, a Swift String is a struct within the `ASwiftClass` struct. You can think of a Swift string as sort of a facade design pattern that hides different types of Swift string types based upon if they are hardcoded, Cocoa, use ASCII, etc. To make it even more interesting, the types and layout will differ if Swift is compiled for a 32-bit or 64-bit platform. For the sake of simplicity, only the 64-bit platform will be discussed.

For 64-bit platforms, the memory layout of a Swift String comprises 16 bytes with the structural layout depending on the type of String. That is, you will need to first determine the String's type before you can correctly parse the String's contents.

So how can one determine the type? By consulting the documentation available on the Swift repo!

This documentation is taken from the https://github.com/apple/swift/blob/master/stdlib/public/core/StringObject.swift for Swift 4.2

```
// ## _StringObject bit layout
//
// x86-64 and arm64: (one 64-bit word)
// +---+---+---|---+-----+-----------------------------------------+
// + t | v | o | w | uuuu | payload (56 bits)                       |
// +---+---+---|---+-----+-----------------------------------------+
// most significant bit                          least significatn bit
//
// where t: is-a-value, i.e. a tag bit that says not to perform ARC
//       v: sub-variant bit, i.e. set for isCocoa or isSmall
//       o: is-opaque, i.e. opaque vs contiguously stored strings
//       w: width indicator bit (0: ASCII, 1: UTF-16)
```

```
//       u: unused bits
//
// payload is:
//   isNative: the native StringStorage object
//   isCocoa: the Cocoa object
//   isOpaque & !isCocoa: the _OpaqueString object
//   isUnmanaged: the pointer to code units
//   isSmall: opaque bits used for inline storage // TODO: use them!
//
```

From the documentation, the **t**, **v**, **o**, **w** bits are used to help determine the type of Swift String. The following 4 **u** bits will be used by the specific string type. That is, the first 4 bits of the **_object** variable for the **_StringCore** struct mentioned above will give you this information.

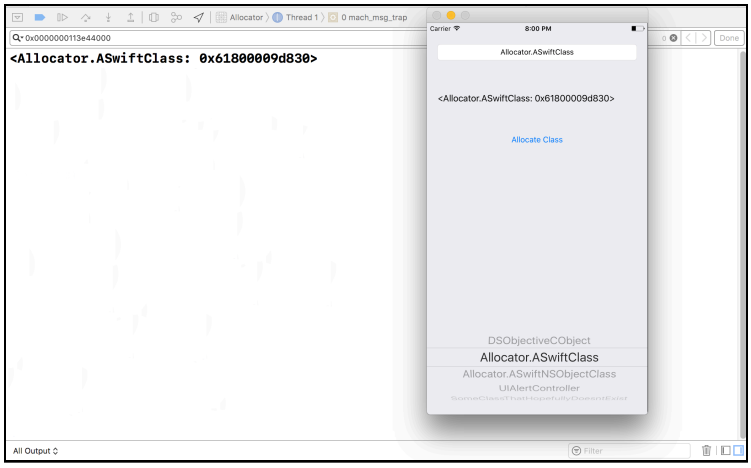
The layout of the Swift String struct makes the assembly calling convention rather interesting. If you pass a String to a function, it will actually pass in two parameters (and use two registers) instead of a pointer to a struct containing the two parameters (in one register). Don't believe me? Check it out yourself when you're done with this chapter!

Back to LLDB and jumping through an object.

Clear the LLDB screen with a ⌘ + **K**, then resume the application through LLDB or the Xcode GUI.

You're going to do the exact same thing with the `ASwiftClass` that you did with `DSObjectiveCObject`. Use the developer/designer "approved" `UIPickerView` and select **Allocator.SwiftClass**. Remember, to correctly reference a Swift class (i.e. in `NSClassFromString` and friends), you need the module name prepended to the classname with a period separating the two.

Tap the **Allocate Class** button and copy the memory address spat out to the console.



You'll get something similar to the following:

```
<Allocator.ASwiftClass: 0x61800009d830>
```

Usually, Swift hides the pointer in the `description` and `debugDescription` methods, but there's something sneaky compiled into this project that you'll come across in a second.

For now, grab that memory address and stick it in the clipboard.

First use LLDB to ensure it's valid, by `po`-ing it:

```
(lldb) po 0x61800009d830
```

If you get something different than the following, you should be more than somewhat surprised:

```
<Allocator.ASwiftClass: 0x61800009d830>
```

Even though this is a pure Swift object, you were able to get the dynamic description in the Objective-C context. That means you can climb the class hierarchy to see the parent class!

```
(lldb) po [0x61800009d830 superclass]
```

You'll get an interesting class with the name of:

```
SwiftObject
```

You'll explore this class more in a second. For now, start jumping around in memory. Dereference the pointer's address and prove to yourself that the first parameter is that `isa Class` variable:

```
(lldb) po *(id *)0x61800009d830
```

You'll get `Allocator.ASwiftClass`. Now check out that reference counter variable:

```
(lldb) po *(id *)(0x61800009d830 + 0x8)
```

You'll get something similar to the following:

```
0x0000000000000002
```

It's clear that the address here is not a Objective-C address, since `*(id *)0x0000000200000004` would point to a class if it were a valid instance/class. Instead, this is the reference counter unique to Swift classes. Let's see how this thing works.

Use LLDB to manually retain this class:

```
(lldb) po [0x61800009d830 retain]
```

Press the up arrow twice to retrieve the previous command and execute it again:

```
(lldb) po *(id *)(0x61800009d830 + 0x8)
```

You'll now get a slightly different number:

```
0x0000000200000002
```

Notice the middle hex value jumped up by 2. Shooting from the hip, this giant word should actually be viewed as 2 separate integer (32-bit) fields instead of one 64-bit field. See if `release`'ing this reference brings the count back down:

```
(lldb) po [0x61800009d830 release]
```

Yep, up arrow twice again, then Enter.

```
(lldb) po *(id *)(0x61800009d830 + 0x8)
```

You'll get your happy, original value:

```
0x0000000000000002
```

Now that you've got past the `isa` and the `refCounts` it's time to turn your attention to those lovely properties in the `ASwiftClass` instance.

Clear the screen to start fresh, then increment your offset amount in LLDB.

```
(lldb) po *(id *)(0x61800009d830 + 0x10)
```

You'll get the internal representation of `UIColor`'s brown:

```
UIExtendedSRGBColorSpace 0.6 0.4 0.2 1
```

Jump another 8 bytes and start exploring the `firstName` object's structure. You're going to change around how you view the display of the data. Remember those first 4 bits of a Swift String that determine the type? You will examine those now.

Type the following:

```
(lldb) x/gt '0x61800009d830 + 0x18'
```

You'll get the following:

```
0x600003c3aa18:
0b1110010100000000000000000000000000000000000000000000000000000000
```

Check out those first four bits on the far left. You can tell that:

- bit 0(t): This object is not reference counted with ARC, which explains why the value initially had zero when executing the `retain` method earlier.
- bit 1(v): The variant is set, for this case, the String is internally known as a **Swift small String**. More on that in a second.
- bit 2(o): This instance is stored as an opaque string
- bit 3(w): The value is not set, which means that ASCII is used for this reference.

This String reference is a **small String**, which is a Swift String that takes up less than 15 bytes. That means all of the contents can be referenced inside of the Swift String struct (try saying that 3 times fast). If the string was greater than 15 bytes, a pointer would be needed to reference the data instead of just packing it into the 16 byte struct (15 bytes for data, 1 byte for type and string length).

The layout of the small String types can be found here: https://github.com/apple/swift/blob/master/stdlib/public/core/SmallString.swift

Here is a simplified C layout of the UTF-8 small Swift String:

```
typedef struct {
  char spillover[7];
  char bits; // msb (tvow) bit types, lsb (uuuu) string length
  char start[8]; // start address of String
} SmallUTF8String;
```

In this struct, if a string has a length greater than 8 bytes, the spillover variable is used to start the remaining characters. There's also the `bits` value, which stores the type as well as count on the lower 4 bits.

Use LLDB to explore the layout of the **`firstName`** variable at the `start` offset in LLDB:

```
(lldb) x/s '0x61800009d830 + 0x20'
"Derek"
```

Since the String contains the value "Derek", the `start` offset can be used.

What about the length of the string? You can easily view the data by typing the following:

```
(lldb) x/gx '0x61800009d830 + 0x18'
```

You'll see something like:

```
0x61800009d848: 0xe500000000000000
```

That 5 is the value you want. This is the lower 4 bits in the most significant side (represented by the `bits` variable in the `SmallUTF8String` struct). To truly isolate this you can execute the following:

```
p/d *(int *)(0x61800009d830 + 0x18 + 7) & 0xf
```

This will give the String's length, which has to be 0-15 due to the limiation length of a small Swift string.

No need to go after the **`lastName`** property. You've got the idea of how this works. If you did want to hunt for that on your own time, you would want to start at offset `0x28` of the `ASwiftClass` instance.

Swift memory layout with NSObject superclass

Final one. You know the drill, so we'll speed this one up a bit and skip the actual debugging session.

Check out the sourcecode for **ASwiftNSObjectClass.swift**:

```
class ASwiftNSObjectClass: NSObject {
  let eyeColor = UIColor.brown
  let firstName = "Derek"
  let lastName = "Selander"

  required override init() { }
}
```

It's the same thing as the `ASwiftClass`, except it inherits from `NSObject` instead of from nothing.

So is there any difference in the generated C struct pseudocode?

```
struct ASwiftNSObjectClass {
  Class isa;
  UIColor *eyeColor;

  struct _StringCore {
    uintptr_t _object;
    uintptr_t rawBits;
  } firstName;

  struct _StringCore {
    uintptr_t _object;
    uintptr_t rawBits;
  } lastName;
}
```

Almost! The only difference is that the `ASwiftNSObjectClass` instance is missing the `refCounts` variable at offset `0x8`, the rest of the layout in memory will be the same.

Let's skip the debugging session and just talk about what will happen when you try **retain**'ing an instance of this class: the **refCounts** variable will *not* be modified. This makes sense because Objective-C has its own implementation of `retain/release` that's different from the Swift implementation.

You can finally look at the **`SBValue`** class I've been itching to describe to you!

SBValue

Yay! Time to talk about this awesome class.

`SBValue` is responsible for interpreting the parsed expressions from your JIT code. Think of `SBValue` as a representation that lets you explore the members within your object, just as you did above, but without all that ugly dereferencing. Within the `SBValue` instance, you can easily access all members of your struct... er, I mean, your Objective-C or Swift classes.

Within the `SBTarget` and `SBFrame` class, there's a method named **`EvaluateExpression`**, which will take your expression as a Python `str` and return an `SBValue` instance. In addition, there's an optional second parameter that lets you specify how you want your code to be parsed. You'll start without the optional second parameter, and explore it later.

Jump back into the LLDB console and make sure the Allocator project is still running. Make sure the LLDB console is up (i.e. the program is paused), clear the console and type the following:

```
(lldb) po [DSObjectiveCObject new]
```

You'll get something similar to the following:

```
<DSObjectiveCObject: 0x61800002eec0>
```

This ensures you can create a valid instance of a `DSObjectiveCObject`.

This code works, so you can apply it to the `EvaluateExpression` method of either the global `SBTarget` or `SBFrame` instance:

```
(lldb) script lldb.frame.EvaluateExpression('[DSObjectiveCObject new]')
```

You'll get the usual cryptic output with the class but no context to describe what this does:

```
<lldb.SBValue; proxy of <Swig Object of type 'lldb::SBValue *' at
0x10ac78b10> >
```

You've got to use `print` to get context for these classes:

```
(lldb) script print lldb.target.EvaluateExpression('[DSObjectiveCObject
new]')
```

You'll get your happy `debugDescription` you've become accustomed to.

```
(DSObjectiveCObject *) $2 = 0x0000618000034280
```

> **Note**: If you mistype something, you'll still get an instance of `SBValue`, so make sure it's printed out the item you expect it should. For example, if you mistyped the JIT code, you might get something like `** = <could not resolve type>**` from the `SBValue`.
>
> You can verify the `SBValue` succeeded by checking the **`SBError`** instance within your `SBValue`. If your `SBvalue` was named `sbval`, you could do `sbval.GetError().Success()`, or more simply `sbval.error.success`. `print` as a quick way to see if it worked or not.

Modify this command so you're assigning it to the variable **`a`** inside the Python context:

```
(lldb) script a = lldb.target.EvaluateExpression('[DSObjectiveCObject
new]')
```

Now apply the Python `print` function to the `a` variable:

```
(lldb) script print a
```

Again, you'll get something similar to the following:

```
(DSObjectiveCObject *) $0 = 0x0000608000033260
```

Great! You have a `SBValue` instance stored at `a` and are already knowledgeable about the memory layout of the `DSObjectiveCObject`. You know `a` is holding a `SBValue` that is a pointer to the `DSObjectiveCObject` class.

You can grab the `description` of the `DSObjectiveCObject` class by using the `GetDescription()`, or more simply `description` property of `SBValue`.

Type the following:

```
(lldb) script print a.description
```

You'll see something similar to the following:

```
<DSObjectiveCObject: 0x608000033260>
```

You can also get the `value` property, which returns a Python `String` containing the address of this instance:

```
(lldb) script print a.value
```

Just the value this time:

```
0x0000608000033260
```

Copy the output of `a.value` and ensure `po`'ing this pointer gives you the original, correct reference:

```
(lldb) po 0x0000608000033260
```

Yup:

```
<DSObjectiveCObject: 0x608000033260>
```

If you want the address expressed in a Python number instead of a Python `str`, you can use the `signed` or `unsigned` property:

```
(lldb) script print a.signed
```

Like this:

```
106102872289888
```

Formatting the number to hexadecimal will produce the pointer to this instance of `DSObjectiveCObject`:

```
(lldb) p/x 106102872289888
```

And you're back to where you were before:

```
(long) $3 = 0x0000608000033260
```

Exploring properties through SBValue offsets

What about those properties stuffed inside that `DSObjectiveCObject` instance? Let's explore those!

Use the **GetNumChildren** method available to SBValue to get its child count:

```
(lldb) script print a.GetNumChildren()
```

You'll get 4 (or potentially 3 depending on the version of LLDB/run conditions which hides an instance's isa variable)

You can think children as just an array. There's a special API to traverse the children in a class called **GetChildAtIndex**, so you can explore children 0-3 in LLDB.

Child 0:

```
(lldb) script print a.GetChildAtIndex(0)
(NSObject) NSObject = {
  isa = DSObjectiveCObject
}
```

Child 1:

```
(lldb) script print a.GetChildAtIndex(1)
(UICachedDeviceRGBColor *) _eyeColor = 0x0000608000070e00
```

Child 2:

```
(lldb) script print a.GetChildAtIndex(2)
(__NSCFConstantString *) _firstName = 0x000000010db83368 @"Derek"
```

Child 3:

```
(lldb) script print a.GetChildAtIndex(3)
(__NSCFConstantString *) _lastName = 0x000000010db83388 @"Selander"
```

Each of these will return a SBValue in itself, so you can explore that object even further if you desired. Take the firstName property into account. Type the following to just get the description:

```
(lldb) script print a.GetChildAtIndex(2).description
Derek
```

It's important to remember the Python variable a is a pointer to an object. Type the following:

```
(lldb) script a.size
8
```

This will print out a value saying `a` is 8 bytes long. But you want to get to the actual content! Fortunately, the `SBValue` has a **deref** property that returns another `SBValue`. Explore the output with the `size` property:

```
(lldb) script a.deref.size
```

This returns the value **32** since it makes up the `isa`, `eyeColor`, `firstName`, and `lastName`, each of them being 8 bytes long themselves as they are all pointers.

Here's another way to look at what the `deref` property is doing. Explore the `SBType` class (you can look that one up yourself) of the `SBValue`.

```
(lldb) script print a.type.name
```

You'll get this:

```
DSObjectiveCObject *
```

Now do the same thing through the **deref** property:

```
(lldb) script print a.deref.type.name
```

You'll now get the normal class:

```
DSObjectiveCObject
```

Viewing raw data through SBValue

You can even dump the raw data out with the **`data`** property in `SBValue`! This is represented by a class named **`SBData`**, which is yet another class you can check out on your own.

Print out the data of the pointer to `DSObjectiveCObject`:

```
(lldb) script print a.data
```

This will print out the physical bytes that make up the object. Again, this is the pointer to `DSObjectiveCObject`, not the object itself.

```
60 32 03 00 80 60 00 00                          `2...`..
```

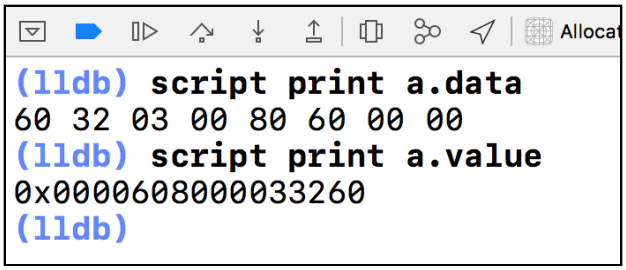
Remember, each byte can be represented as two digits in hexadecimal.

Do you remember covering the **little-endian** formatting in Chapter 12, "Assembly & Memory," and how the raw data is reversed?

Compare this with the **`value`** property of `SBValue`.

```
(lldb) script print a.value
```

Which will give you the expected `0x0000608000033260`



Notice how the values have been flipped. For example, the final two hex digits of my pointer are the first grouping (aka byte) in the raw data. In my case, the raw data contains `0x60` as the first value, while the pointer contains `0x60` as the final value.

Use the **`deref`** property to grab *all* the bytes that make up this `DSObjectiveCObject`.

```
(lldb) script print a.deref.data
f0 54 b8 0d 01 00 00 00 0e 07 00 80 60 00 00  .T..........`..
68 33 b8 0d 01 00 00 00 88 33 b8 0d 01 00 00 00  h3.......3......
```

This is yet another way to visualize what is happening. You were jumping 8 bytes each time when you were spelunking in memory with that cute `po *(id*)(0x0000608000033260 + multiple_of_8)` command.

SBExpressionOptions

As mentioned when discussing the `EvaluateExpression` API, there's an optional second parameter that will take an instance of type **SBExpressionOptions**. You can use this command to pass in specific options for the JIT execution.

In LLDB, clear the screen, start fresh and type the following:

```
(lldb) script options = lldb.SBExpressionOptions()
```

You'll get no output upon success. Next, type:

```
(lldb) script options.SetLanguage(lldb.eLanguageTypeSwift)
```

`SBExpressionOptions` has a method named **`SetLanguage`** (when in doubt, use **`gdocumentation SBExpressionOptions`**), which takes an LLDB module enum of type **lldb::LanguageType**. The LLDB authors have a convention for sticking an "e" before an enum, the enum name, then the unique value.

This sets the options to evaluate the code as Swift instead of whatever the default is, based on the language type of `SBFrame`.

Now tell the `options` variable to interpret the JIT code as a of type ID (i.e. `po`, instead of `p`):

```
(lldb) script options.SetCoerceResultToId()
```

`SetCoerceResultToId` takes an optional Boolean, which determines if it should be interpreted as an `id` or not. By default, this is set to `True`.

To recap what you did here: you set the options to parse this expression using the Python API instead of the options passed to us through the expression command.

For example, `SBExpressionOptions` you've declared so far is pretty much equivalent to the following options in the `expression` command:

```
expression -lswift -O -- your_expression_here
```

Next, create an instance of the `ASwiftClass` method only using the `expression` command. If this works, you'll try out the same expression in the `EvaluateExpression` command. In LLDB type the following:

```
(lldb) e -lswift -O -- ASwiftClass()
```

You'll get an ugly little error for output…

```
error: <EXPR>:3:1: error: use of unresolved identifier 'ASwiftClass'
ASwiftClass()
^~~~~~~~~~
```

Oh yeah, — you need to import the **Allocator** module to make Swift play nicely in the debugger.

In LLDB:

```
(lldb) e -lswift -- import Allocator
```

> **Note**: This is a problem many LLDB users complain about: LLDB can't properly evaluate code that should be able to execute. Adding this import logic will modify LLDB's Swift **expression prefix**, which is basically a set of header files that a referenced right before you JIT code is evaluated.
>
> LLDB can't see the class `ASwiftClass` in the JIT code when you're stopped in the non-Swift debugging context. This means you need to append the headers to the expression prefix that belongs to the Allocator module.

There's a great explanation from one of the LLDB authors about this very problem here: http://stackoverflow.com/questions/19339493/why-cant-lldb-evaluate-this-expression.

Execute the previous command again. Up arrow twice then Enter:

```
(lldb) e -lswift -O -- ASwiftClass()
```

You'll get a reference to an instance of the `ASwiftClass()`.

Now that you know this works, use the **`EvaluateExpression`** method with the `options` parameter as the second parameter this time and assign the output to the variable **`b`**, like so:

```
(lldb) script b = lldb.target.EvaluateExpression('ASwiftClass()',
options)
```

If everything went well, you'll get a reference to a `SBValue` in the `b` Python variable.

Note: It's worth pointing out some properties of `SBValue` will not play nicely with Swift. For example, dereferencing a Swift object with `SBValue`'s `deref` or `address_of` property will not work properly. You can coerce this pointer to an Objective-C reference by casting the pointer as a `SwiftObject`, and everything will then work fine. Like I said, they make you work for it when you're trying to go after pointers in Swift!

Referencing variables by name with SBValue

Referencing child `SBValues` via **`GetChildAtIndex`** from `SBValue` is a rather ho-hum way to navigate to an object in memory. What if the author of this class added a property before `eyeColor` that totally screwed up your offset logic when traversing this `SBValue`?

Fortunately, `SBValue` has yet *another* method that lets you reference instance variables by name instead of by offset: **`GetValueForExpressionPath`**.

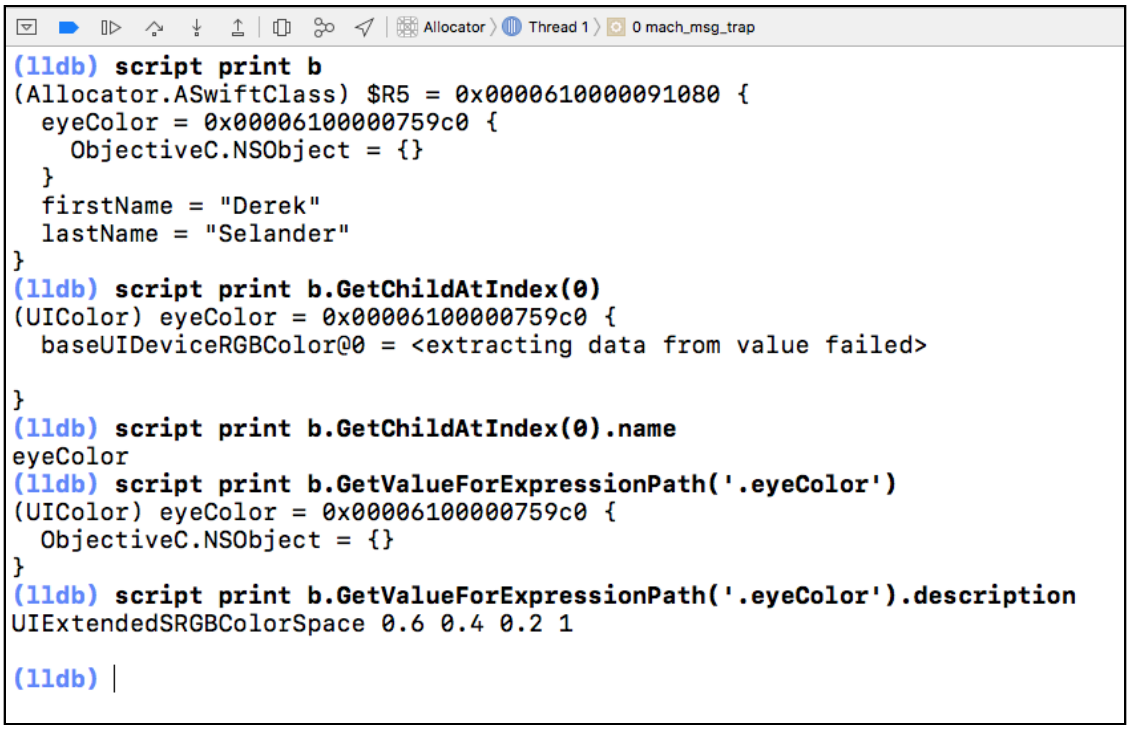
Jump back to LLDB and type the following:

```
(lldb) script print b.GetValueForExpressionPath('.firstName')
```

You can keep drilling down into the child's own struct if you wish:

How did I obtain the name of child `SBValues`? If you had no clue of the name for the child `SBValue`, all you have to do is get to the child using the **`GetIndexOfChild`** API, then use the **`name`** property on that `SBValue` child.

For example, if I didn't know the name of the `UIColor` property found in the **`b`** `SBValue`, I could do the following:



lldb.value

One final cool thing you can do is create a Python reference that contains the `SBValue`'s properties as the Python object's properties (wait... what?). Think of this as an object through which you can reference variables using Python properties instead of Strings.

Back in the console, instantiate a new **`value`** object from your **`b`** `SBValue`:

```
(lldb) script c = lldb.value(b)
```

This will create the special LLDB Python object of type `value`. Now you can reference its instance variables just like you would a normal object!

Type the following into LLDB:

```
(lldb) script print c.firstName
```

You can also cast the child object back into a `SBValue` so you can query it or apply it to a `for` loop, like so:

```
(lldb) script print c.firstName.sbvalue.signed
```

Again, if you don't know the name of a child `SBValue`, use the `GetChildAtIndex` API to get the child and get its name from the `name` property.

> **Note**: Although the `lldb.value` class is awesome, this comes at a cost. It's rather expensive to create and access properties through this type of class. If you are parsing a huge `NSArray` (or `Array<Any>`, for you Swifties), using this class will definitely slow you down. Play around with it and find the sweet spot between speed and convenience.

Where to go from here?

Holy cow… how dense was that chapter!? Fortunately you have come full circle. You can use the options provided by your custom command to dynamically generate your JIT script code. From the return value of your JIT code, you can write scripts that have custom logic based upon the return `SBValue` that is parsed through the `EvaluateExpression` APIs.

This can unlock some amazing scripts for you. In any process to which you can attach LLDB, you can run your own custom code and handle your own custom return values within your Python script. There's no need to deal with signing issues or loading of frameworks or anything like that.

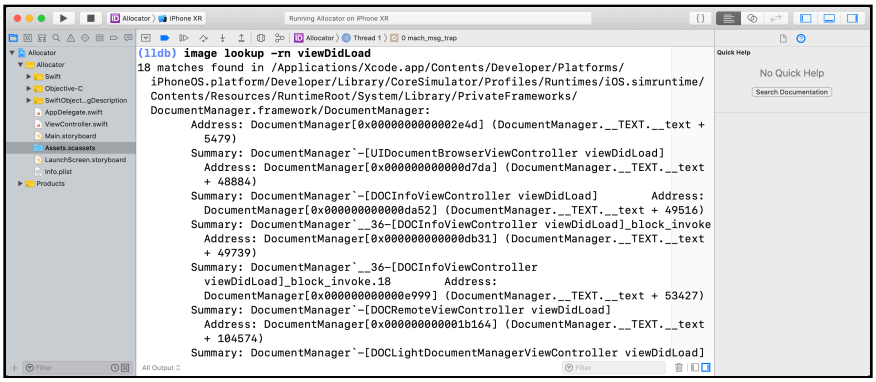
The remaining chapters in this section will focus on the composition of some creative scripts and how they can make your debugging (or reverse engineering) life much simpler. Theory time is over. It's time for some fun!

Chapter 26: SB Examples, Improved Lookup

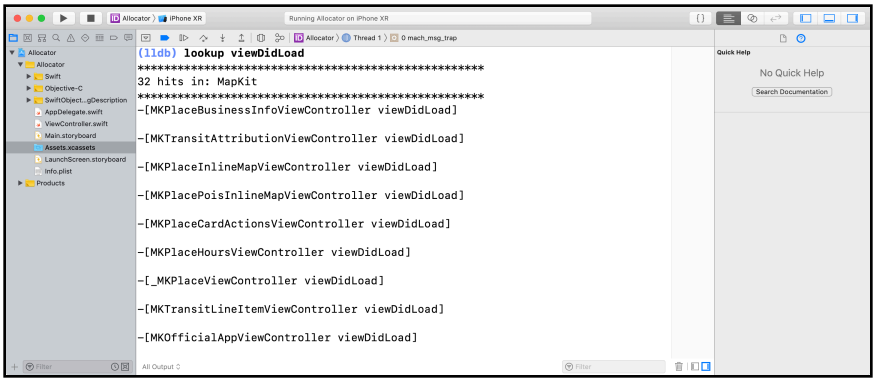
For the rest of the chapters in this section, you'll focus on Python scripts.

As alluded to in the previous chapter, the `image lookup -rn` command is on its way out. Time to make a prettier script to display content.

Here's what you get right now with the `image lookup -rn` command:



When you finish this chapter, you'll have a new script named **`lookup`** which queries in a *much* cleaner way.



In addition, you'll add a couple of parameters to the `lookup` command to add some bells and whistles for your new searches.

Automating script creation

Included with the **starter** directory of this project are two Python scripts that will make your life easier when creating LLDB script content. They are as follows:

- **generate_new_script.py**: This will create a new skeleton script with whatever name you provide it and stick it into the same directory `generate_new_script` resides in.
- **lldbinit.py**: This script will enumerate all scripts (files that end with `.py`) located within the same directory as itself and try to load them into LLDB. In addition, if there are any files with a `txt` extension, LLDB will try to load those files' contents through `command import`.

Take both of these files found in the **starter** folder of this chapter and stick them into your **~/lldb/** directory.

Once the files are in their correct locations, jump over to your **~/.lldbinit** file and add following line of code:

```
command script import ~/lldb/lldbinit.py
```

This will load the **lldbinit.py** file which will enumerate all `.py` files and `.txt` files found in the same directory and load them into LLDB. This means that from here on out, simply adding a script file into the `~/lldb` directory will load it automatically once LLDB starts.

Creating the lookup command

With your new tools properly set up, open up a Terminal window. Launch a new instance of LLDB:

```
lldb
```

As expected, you'll be greeted by the LLDB prompt.

Make sure there are no build errors in any of your existing LLDB scripts:

```
(lldb) reload_script
```

If your output is free of errors, it's time to try out your new command **__generate_script** (implemented from the **generate_new_script.py** file).

In LLDB, type:

```
(lldb) __generate_script lookup
```

If everything went as expected, you'll get output similar to the following:

```
Opening "/Users/derekselander/lldb/lookup.py"...
```

In addition, a Finder window will pop up showing you the location of the file. It's pretty crazy what you can do with these Python scripts, right?

Hold onto the Finder window for a second — don't close it. Head back to the LLDB Terminal window and apply the **reload_script** command.

Since the `lookup.py` script was created in the same directory as the `lldbinit.py` file and you have just reloaded the contents of `~/.lldbinit`, you'll now have a working skeleton of the lookup.py file. Give the command a go.

```
(lldb) lookup
```

You'll get the following output:

```
Hello! the lookup command is working!
```

Now you can create and use custom commands in as little as two LLDB commands. Yeah, you could do all the setup in one command, but I like having control over when my scripts reload.

lldbinit directory structure suggestions

The way I've structured my own `lldbinit` files might be insightful to some. This is not a required section, but more of a suggestion on how to organize all of your custom scripts and content for LLDB.

I tend to keep my `~/.lldbinit` as light as possible and use a script like `lldbinit.py` to load all my contents from a particular directory. Facebook's Chisel does the same thing with the **fblldb.py** file. Check it out if you're interested.

I keep that directory under source control in case I need to transfer logic to a different computer, or in case I completely screw something up. For example, my actual `~/.lldbinit` file (when not working on this book) only contains the following items:

```
command script import /Users/derekselander/lldb_repo/lldb_commands/
lldbinit.py
command script import /Users/derekselander/chisel/chisel/fblldb.py
```

The `lldb_repo` is a public git repository at https://github.com/DerekSelander/lldb which contains some LLDB scripts designed for reverse engineering.

I also have Facebook's Chisel on source control, so whenever those developers push a new, interesting release, I'll just pull the latest from my Chisel source control directory at https://github.com/facebook/chisel and I'll have everything I need the next time I run LLDB, or reload my scripts through `reload_script`.

Inside my `lldb_commands` directory, I have all my Python scripts as well as two text files. One text file is named **cmds.txt** and holds all my `command regex`'s and `command alias`'s. I also have another file named **settings.txt**, which I use to augment any LLDB settings.

For example, the only content I have in my `settings.txt` file at the moment is:

```
settings set target.skip-prologue false
settings set target.x86-disassembly-flavor intel
```

You've already added these settings to your `~/.lldbinit` file earlier in this book, but I prefer this implementation to separate out my custom LLDB commands to my LLDB settings so I don't get lost when `grep`'ing my `~/.lldbinit` file.

However, for this book, I chose to keep each chapter content independent for each script installation. This means you've manually added content to your `~/.lldbinit` file so you know what's happening. You should revisit this new structure implementation when (if?) you finish this book, as there are several benefits to this suggested layout. The benefits are as follows:

1. Calling `reload_script` only displays the commands `~/.lldbinit` is loading; it will not display the sub-scripts being loaded. For example this will echo back the `lldbinit.py` being loaded, but not echo out the content `lldbinit.py` itself loads.

   This makes it easier to create scripts because I often use `reload_script` as a way to check for any error messages on the latest script I am working on. The less output there is from executing `reload_script`, the less output there is to review when checking for errors in the console.

2. As noted, having as little content as possible in `~/.lldbinit` will let you easily transfer content between computers, especially if that content is under source control.

3. Finally, it's much easier to add new scripts with this implementation. Just stick them in the same directory as the `lldbinit.py` file and it will be loaded next time. The alternative is to manually add the path to your script to the `~/.lldbinit` file, which can get annoying if you do this frequently.

That's my two cents on the subject. You'll use this implementation strategy for the remaining scripts in this section as you only have to add scripts to your `~/lldb` directory for them to get loaded into LLDB… which is rather nice, right?

Back to the `lookup` command!

Implementing the lookup command

As you saw briefly in the previous chapter, the foundation behind this `lookup` command is rather simple. The main "secret" is using `SBTarget`'s **`FindGlobalFunctions`** API. After that, all you need to do is format the output as you like.

You'll continue working with the **Allocator** Xcode project, found in the **starter** folder for this chapter.

Open the project, and build and run on a iPhone XS Simulator. You'll use this project to test out your new `lookup` command queries as the script progresses throughout the chapter.

Once running, pause the application and bring up LLDB.

My memory is a little fuzzy. Which parameters does this `FindGlobalFunctions` specify? Type the following into LLDB:

```
(lldb) script help(lldb.SBTarget.FindGlobalFunctions)
```

You'll get the following output showing the method signature:

```
FindGlobalFunctions(self, *args) unbound lldb.SBTarget method
    FindGlobalFunctions(self, str name, uint32_t max_matches, MatchType
matchtype) -> SBSymbolContextList
```

Since it's a Python class, you can ignore that first **`self`** parameter. The `str` parameter named **name** will be your lookup query. **`max_matches`** will dictate the maximum number of hits you want. If you specify the number 0, it will return all available matches. The **`matchType`** parameter is a lldb Python enum on which you can perform different types of searches, such as regex or non-regex.

Since regex searching really is the only way to go, you'll use the LLDB enum value **`lldb.eMatchTypeRegex`**.

The other enum values can be found here: https://lldb.llvm.org/python_reference/lldb%27-module.html#eMatchTypeRegex

Time to implement this in the **lookup.py** script. Open up **~/lldb/lookup.py** in your favorite text editor. Find the following code at the end of **handle_command**:

```
# Uncomment if you are expecting at least one argument
# clean_command = shlex.split(args[0])[0]
result.AppendMessage('Hello! the lookup command is working!')
```

Delete the above code, and replace it with the following, making sure you preserve the indentation:

```
# 1
clean_command = shlex.split(args[0])[0]
# 2
target = debugger.GetSelectedTarget()

# 3
contextlist = target.FindGlobalFunctions(clean_command, 0,
lldb.eMatchTypeRegex)
# 4
result.AppendMessage(str(contextlist))
```

Here's what this does:

1. Obtains a cleaned version of the command that was passed to the script, using the same magic as you saw in Chapter 24.
2. Grabs the instance of `SBTarget` through `SBDebugger`.
3. Uses the `FindGlobalFunctions` API with `clean_command`. You're supplying 0, for no upper limit on number of results and giving it the `eMatchTypeRegex` match type to use a regular expression search.
4. You're turning the `contextlist` into a Python `str` and then appending it to the `SBCommandReturnObject`.

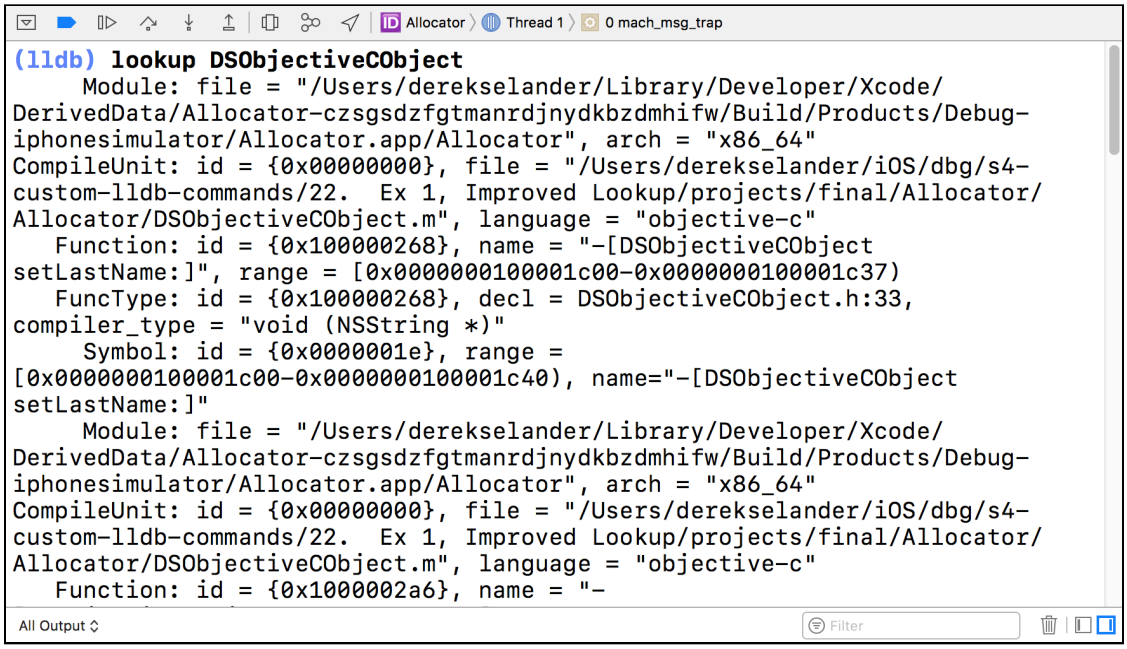
Back in Xcode, reload the contents through the LLDB console:

```
(lldb) reload_script
```

Give the `lookup` command a go. Remember that `DSObjectiveCObject` class you spelunked in the previous chapter? Dump everything pertaining to that through LLDB:

```
lookup DSObjectiveCObject
```

You'll get output that actually looks worse than **`image lookup -rn DSObjectiveCObject`**:



Use LLDB's **`script`** command to figure out which APIs to explore further:

```
(lldb) script k = lldb.target.FindGlobalFunctions('DSObjectiveCObject',
0, lldb.eMatchTypeRegex)
```

This will replicate what you've done in the `lookup.py` script and assign the instance of `SBSymbolContextList` to the value `k`. I am a fan of short variables names when exploring API names — if you haven't noticed.

Explore the documentation of `SBSymbolContextList`:

```
(lldb) gdocumentation SBSymbolContextList
```

While you're at it, dump all the all the methods implemented by `SBSymbolContextList`. In LLDB:

```
(lldb) script dir(lldb.SBSymbolContextList)
```

This will dump out all the methods `SBSymbolContextList` implements or overrides. There's a lot there. But focus on the **`__iter__`** and the **`__getitem__`**.

```
(lldb) script dir(lldb.SBSymbolContextList)
['Append', 'Clear', 'GetContextAtIndex', 'GetDescription', 'GetSize',
'IsValid', '__class__', '__del__', '__delattr__', '__dict__', '__doc__',
'__format__', '__getattr__', '__getattribute__', '__getitem__', '__hash__',
'__init__', '__iter__', '__len__', '__module__', '__new__', '__nonzero__',
'__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__',
'__str__', '__subclasshook__', '__swig_destroy__', '__swig_getmethods__',
'__swig_setmethods__', '__weakref__', 'blocks', 'compile_units', 'functions',
'get_block_array', 'get_compile_unit_array', 'get_function_array',
'get_line_entry_array', 'get_module_array', 'get_symbol_array',
'line_entries', 'modules', 'symbols']
(lldb)
```

This is good news for your script, since this means `SBSymbolContextList` is **iterable** as well as **indexable**. A second ago, you just assigned an instance of `SBSymbolContextList` to a variable named `k` through LLDB.

In the LLDB console, use indexing to grab an item in the `k` object.

```
(lldb) script k[0]
```

This is equivalent to (though much more ugly) typing `script k.__getitem__(0)`. You'll get something like:

```
<lldb.SBSymbolContext; proxy of <Swig Object of type
'lldb::SBSymbolContext *' at 0x113a83780> >
```

Good to know! The `SBSymbolContextList` holds an "array" of `SBSymbolContext`.

Use the `print` command to get the context of this `SBSymbolContext`:

```
(lldb) script print k[0]
```

Your output could differ, but I got the `SBSymbolContext` which represents `-[DSObjectiveCObject setLastName:]`, like so:

```
     Module: file = "/Users/derekselander/Library/Developer/Xcode/
DerivedData/Allocator-czsgsdzfgtmanrdjnydkbzdmhifw/Build/Products/Debug-
iphonesimulator/Allocator.app/Allocator", arch = "x86_64"
CompileUnit: id = {0x00000000}, file = "/Users/derekselander/iOS/dbg/s4-
custom-lldb-commands/22.  Ex 1, Improved Lookup/projects/final/Allocator/
Allocator/DSObjectiveCObject.m", language = "objective-c"
   Function: id = {0x100000268}, name = "-[DSObjectiveCObject
setLastName:]", range = [0x0000000100001c00-0x0000000100001c37)
   FuncType: id = {0x100000268}, decl = DSObjectiveCObject.h:33,
compiler_type = "void (NSString *)"
     Symbol: id = {0x0000001e}, range =
[0x0000000100001c00-0x0000000100001c40), name="-[DSObjectiveCObject
setLastName:]"
```

You'll use properties and/or getter methods from the `SBSymbolContext` to grab the name of this function.

The easiest way to do this is to grab the **`SBSymbol`** from the `SBSymbolContext` through the **`symbol`** property. From there the `SBSymbol` contains a **name** property, which will return your happy Python string.

Make sure this works in your LLDB console:

```
(lldb) script print k[0].symbol.name
```

In my case, I received the following:

```
-[DSObjectiveCObject setLastName:]
```

This is enough information to work with in building out your script. You'll take the `SBSymbolContextList`, iterate through the items and print out the name of the function it finds.

Head back over to your **lookup.py** script and modify the contents in the **`handle_command`** function. Find the following lines:

```
# 3
contextlist = target.FindGlobalFunctions(clean_command, 0,
lldb.eMatchTypeRegex)
# 4
result.AppendMessage(str(contextlist))
```

Replace them with the following (indenting correctly!):

```
contextlist = target.FindGlobalFunctions(clean_command, 0,
lldb.eMatchTypeRegex)

output = ''
for context in contextlist:
    output += context.symbol.name + '\n\n'

result.AppendMessage(output)
```

You're now iterating all `SBSymbolContext`'s within the returned `SBSymbolContextList`, hunting down the name of the function and separating it by two newlines.

Jump back to Xcode, and reload your script:

```
(lldb) reload_script
```

Then give your updated `lookup` command a test in LLDB:

```
(lldb) lookup DSObjectiveCObject
```

You'll get much prettier output than before:

```
-[DSObjectiveCObject setLastName:]
-[DSObjectiveCObject .cxx_destruct]
-[DSObjectiveCObject setFirstName:]
-[DSObjectiveCObject eyeColor]
-[DSObjectiveCObject init]
-[DSObjectiveCObject lastName]
-[DSObjectiveCObject setEyeColor:]
-[DSObjectiveCObject firstName]
```

This is nice and all, but I want to see where these functions reside in my process. I want to group all functions to a particular module (an **`SBModule`**) when they're being printed out separated by a header with the module name and number of hits for the module.

Head on back to the **lookup.py** file. You'll now create two new functions.

The first function will be named **`generateFunctionDictionary`**, which will take your `SBBreakpointContextList` and generate a Python Dictionary of lists. This `dict` will contain keys for each module. For the value in the dict, you'll have a Python list for each `SBSymbolContext` that gets hit.

The second function will be named **`generateOutput`**, which will parse this dictionary you've created along with the `options` you've received from the `OptionParser` instance. This method will return a String to be printed back to the console.

Start by implementing the **`generateModuleDictionary`** function right below the `handle_command` function in your **lookup.py** script:

```
def generateModuleDictionary(contextlist):
    mdict = {}
    for context in contextlist:
        # 1
        key = context.module.file.fullpath
        # 2
        if not key in mdict:
            mdict[key] = []

        # 3
        mdict[key].append(context)
    return mdict
```

Here's what going on:

1. From within the `SBSymbolContext`, you're grabbing the `SBModule` (**`module`**), then the `SBFileSpec` (**file**), then the Python string of the **`fullPath`** and assigning it to a variable named **`key`**. It's important to grab the `fullPath` (instead of, say, `SBFileSpec`'s **`basename`** property, since there could be multiple modules with the same basename).

2. This `mdict` variable is going to hold a list of all symbols found, split by module. The key in this dictionary will be the module name, and the value will be an array of symbols found in that module. On this line, you're checking if the dictionary already contains a list for this module. If not, a blank list is added for this module key.

3. You're adding the `SBSymbolContext` instance to the appropriate list for this module. You can safely assume that for every key in the `mdict` variable, there will be at least one or more `SBSymbolContext` instances.

> **Note**: A much easier way of getting a unique key would be to just use the **`__str__()`** method `SBModule` has (and pretty much every class in the LLDB Python module). This is the function that gets called when you call Python's **`print`** on one of these objects. However, you wouldn't be learning about all these classes, properties and methods in the process if you just relied on the `__str__()` method.

Right below the `generateModuleDictionary` function, implement the **`generateOutput`** function:

```
def generateOutput(mdict, options, target):
    # 1
    output = ''
    separator = '*' * 60 + '\n'
    # 2
    for key in mdict:
        # 3
        count = len(mdict[key])
        firstItem = mdict[key][0]
        # 4
        moduleName = firstItem.module.file.basename
        output += '{0}{1} hits in {2}\n{0}'.format(separator,
                                                   count,
                                                   moduleName)

        # 5
        for context in mdict[key]:
            query = ''
            query += context.symbol.name
            query += '\n\n'
            output += query
  return output
```

Here's what this does:

1. The **`output`** variable will be the return string that contains all the content eventually passed to your `SBCommandReturnObject`.
2. Enumerate all the keys found in the `mdict` dictionary.
3. This will grab the count for the array and the very first item in the list. You'll use this information to query the module name later.
4. You're grabbing the module name to use in the header output for each section.
5. This will iterate all the `SBSymbolContext` items in the Python `list` and add the names to the `output` variable.

One final tweak before you can test this out.

Augment the code in the **handle_command** function so it utilizes the two new methods you've just created. Find the following code:

```
output = ''
for context in contextlist:
    output += context.symbol.name + '\n\n'
```

And replace it with the following:

```
mdict = generateModuleDictionary(contextlist)
output = generateOutput(mdict, options, target)
```

You know what to do. Go to Xcode; reload contents in LLDB.

```
(lldb) reload_script
```

Check out your new and improved lookup command:

```
(lldb) lookup DSObjectiveCObject
```

You'll get something like this:

```
************************************************************
8 hits in Allocator
************************************************************
-[DSObjectiveCObject setLastName:]

-[DSObjectiveCObject .cxx_destruct]

-[DSObjectiveCObject setFirstName:]

-[DSObjectiveCObject eyeColor]

-[DSObjectiveCObject init]
```

```
-[DSObjectiveCObject lastName]
-[DSObjectiveCObject setEyeColor:]
-[DSObjectiveCObject firstName]
```

Cool. Go after all Objective-C methods that begin with **`initWith`**, and only contain two parameters.

```
(lldb) lookup initWith(\w+\:){2,2}\]
```

You'll get hits from both public and private modules, all loaded into the Allocator process.

Adding options to lookup

You'll keep the options nice and simple and implement only two options that don't require any extra parameters.

You'll implement the following:

- Add load addresses to each query. This is ideal if you want to know where the actual function is in memory.
- Provide a module summary only. Don't produce function names, only list the count of hits per module



The `__generate_script` command added some placeholders for the **`generateOptionParser`** method found at the bottom of the **lookup.py** file. In the **`generateOptionParser`** function, change the function so it contains the following code:

```
def generateOptionParser():
    usage = "usage: %prog [options] code_to_query"
    parser = optparse.OptionParser(usage=usage, prog="lookup")
```

```
    parser.add_option("-l", "--load_address",
          action="store_true",
          default=False,
          dest="load_address",
          help="Show the load addresses for a particular hit")

    parser.add_option("-s", "--module_summary",
          action="store_true",
          default=False,
          dest="module_summary",
          help="Only show the amount of queries in the module")
    return parser
```

There's no need to take a deep dive in this code since you learned about this in a previous chapter. You're creating two supported options, **`-s`**, or **`--module_summary`** and **`-l`**, or **`--load_address`**.

You'll implement the load address option first. In the **`generateOutput`** function, navigate to the for-loop iterating over the `SBSymbolContext`, which starts with the `for context in mdict[key]:` line of code.

Make that for-loop look like this:

```
for context in mdict[key]:
    query = ''

    # 1
    if options.load_address:
        # 2
        start = context.symbol.addr.GetLoadAddress(target)
        end = context.symbol.end_addr.GetLoadAddress(target)
        # 3
        startHex = '0x' + format(start, '012x')
        endHex = '0x' + format(end, '012x')
        query += '[{}-{}]\n'.format(startHex, endHex)

    query += context.symbol.name
    query += '\n\n'
    output += query
```

Here's what that does:

1. You're adding the conditional to see if the `load_address` option is set. If so, this will add content to the output.
2. This traverses the `SBSymbolContext` to the `SBSymbol` (**`symbol`** property) to the `SBAddress` (**`addr`** or **`end\_addr`**) and gets a Python `long` through the **`GetLoadAddress`** method.

There's actually a `load_addr` available to `SBAddress`, but I've found it to be a bit buggy at times, so I've defaulted to using the `GetLoadAddress` API instead. This method expects the `SBTarget` as an input parameter.

3. After you have the `start` and `end` addresses expressed in Python `long`'s, you are formatting them to look pretty and consistent using the Python **format** function.

This pads the number with zeros if needed, notes it should be 12 digits long, and formats it in hexadecimal.

Save your work and revisit Xcode and the LLDB console. Reload.

```
(lldb) reload_script
```

Give your new option a go:

```
(lldb) lookup -l DSObjectiveCObject
```

You'll get output similar to the truncated output:

```
**********************************************************************
8 hits in Allocator
**********************************************************************
[0x0001099d2c00-0x0001099d2c40]
-[DSObjectiveCObject setLastName:]

[0x0001099d2c40-0x0001099d2cae]
-[DSObjectiveCObject .cxx_destruct]
```

Put a breakpoint at an address from this list to see if it matches with the function. Do it like so, replacing the address with one from your list:

```
(lldb) b 0x0001099d2c00
Breakpoint 3: where = Allocator`-[DSObjectiveCObject setLastName:] at
DSObjectiveCObject.h:33, address = 0x00000001099d2c00
```

Great job! One more option to implement and then you're done!

Revisit the **`generateOutput`** function for the final time. Find the following line:

```
moduleName = firstItem.module.file.basename
```

Add the following code right after that line:

```
if options.module_summary:
    output += '{} hits in {}\n'.format(count, moduleName)
    continue
```

This simply adds the number of hits in each module and skips adding the actual symbols.

That's it. No more code. Save, then head back to Xcode to reload your script:

```
(lldb) reload_script
```

Give your **module_summary** option a go:

```
(lldb) lookup -s viewWillAppear
```

You'll get something similar to this:

```
1 hits in: GLKit
18 hits in: ContactsUI
3 hits in: DocumentManager
8 hits in: MapKit
49 hits in: UIKitCore
4 hits in: Allocator
```

That's it! You're done! You've made a pretty powerful script from scratch. You'll use this script to search for code in future chapters. The summary option is a great tool to have when you're casting a wide search and then want to narrow it down further.

Where to go from here?

There are many more options you could add to this `lookup` command. You could make a **`-S`** or **`-Swift_only`** query by going after `SBSymbolContext`'s `SBFunction` (through the **`function`** property) to access the **`GetLanguage()`** API.

While you're at it, you should also add a **`-m`** or **`--module`** option to filter content to a certain module.

If you want to see what else is possible, check out my implementation of lookup here: https://github.com/DerekSelander/LLDB/blob/master/lldb_commands/lookup.py.

Enjoy adding those options!

Chapter 27: SB Examples, Resymbolicating a Stripped ObjC Binary

This will be a novel example of what you can do with some knowledge of the Objective-C runtime mixed in with knowledge of the lldb Python module.

When LLDB comes up against a stripped executable (an executable devoid of DWARF debugging information), LLDB won't have the symbol information to give you the stack trace.

Instead, LLDB will generate a **synthetic** name for a method it recognizes as a method, but doesn't know what to call it.

Here's an example of a synthetic method created by LLDB on an always fun to explore process...

```
___lldb_unnamed_symbol906$$SpringBoard
```

One strategy to reverse engineer the name of this method is to create a breakpoint on it and explore the registers right at the start of the method.

Using your assembly knowledge of the Objective-C runtime, you know the **`RSI`** register (x64) or the **`X1`** register (ARM64) will contain the Objective-C Selector that holds the name of method. In addition, you also have the **`RDI`** (x64) or **`X0`** (ARM64) register which holds the reference to the instance (or class).

However, as soon as you leave the function prologue, you have no guarantee that either of these registers will contain the values of interest, as they will likely be overwritten. What if a stripped method of interest calls another function? The registers you care about are now lost, as they're set for the parameters for this new function. You need a way to resymbolicate a stack trace without having to rely upon these registers.

In this chapter, you'll build an LLDB script that will resymbolicate stripped Objective-C functions in a stack trace.

```
(lldb) bt
* thread #1, queue = 'com.apple.main-thread', stop reason = breakpoint 1.1
  * frame #0: 0x0000000107f5f594 UIKit`-[UIView initWithFrame:]
    frame #1: 0x0000000105fec4fc ShadesOfRay`___lldb_unnamed_symbol13$$ShadesOfRay + 924
    frame #2: 0x0000000107f74da2 UIKit`-[UIView init] + 62
    frame #3: 0x0000000105febd5f ShadesOfRay`___lldb_unnamed_symbol1$$ShadesOfRay + 79
    frame #4: 0x0000000107664d22 UIKit`-[UIApplication sendAction:to:from:forEvent:] + 83
    frame #5: 0x0000000107a9f5c7 UIKit`-[UIBarButtonItem(UIInternal) _sendAction:withEvent:] + 149
    frame #6: 0x0000000107664d22 UIKit`-[UIApplication sendAction:to:from:forEvent:] + 83
    frame #7: 0x00000001077e925c UIKit`-[UIControl sendAction:to:forEvent:] + 67
    frame #8: 0x00000001077e9577 UIKit`-[UIControl _sendActionsForEvents:withEvent:] + 450
    frame #9: 0x00000001077e96eb UIKit`-[UIControl _sendActionsForEvents:withEvent:] + 822
    frame #10: 0x00000001077e84b2 UIKit`-[UIControl touchesEnded:withEvent:] + 618
```

When you called **bt** for this process, LLDB didn't have the function names for the highlighted methods. You will build a new command named **sbt** that will look for stripped functions and try to resymbolicate them using the Objective-C runtime. By the end of the chapter, your `sbt` command will produce this:

```
(lldb) sbt
frame #0: 0x1076f5694 UIKit`-[UIView initWithFrame:]
frame #1: 0x105fec4fc ShadesOfRay`-[RayView initWithFrame:] + 924
frame #2: 0x1076f4da2 UIKit`-[UIView init] + 62
frame #3: 0x105febd5f ShadesOfRay`-[ViewController generateRayViewTapped:] + 79
frame #4: 0x107664d22 UIKit`-[UIApplication sendAction:to:from:forEvent:] + 83
frame #5: 0x107a9f5c7 UIKit`-[UIBarButtonItem(UIInternal) _sendAction:withEvent:] + 149
frame #6: 0x107664d22 UIKit`-[UIApplication sendAction:to:from:forEvent:] + 83
frame #7: 0x1077e925c UIKit`-[UIControl sendAction:to:forEvent:] + 67
frame #8: 0x1077e9577 UIKit`-[UIControl _sendActionsForEvents:withEvent:] + 450
frame #9: 0x1077e96eb UIKit`-[UIControl _sendActionsForEvents:withEvent:] + 822
frame #10: 0x1077e84b2 UIKit`-[UIControl touchesEnded:withEvent:] + 618
```

Those once stripped-out Objective-C function calls are now resymbolicated. As with any of these scripts, you can run this new `sbt` script on any Objective-C executable provided LLDB can attach to it.

So how are you doing this, exactly?

Let's first discuss how one can go about resymbolicating Objective-C code in a stripped binary with the Objective-C runtime.

The Objective-C runtime can list all classes from a particular image (an image being the main executable, a dynamic library, an `NSBundle`, etc.) provided you have the full path to the image. This can be accomplished through the **objc_copyClassNamesForImage** API.

From there, you can get a list of all classes returned by `objc_copyClassNamesForImage` where you can dump all class and instance methods for a particular class using the **`class_copyMethodList`** API.

Therefore, you can grab all the method addresses and compare them to the addresses of the stack trace. If the stack trace's function can't generate a default function name (such as if the `SBSymbol` is synthetically generated by LLDB), then you can assume LLDB has no debug info for this address.

Using the lldb Python module, you can get the starting address for a particular function — even when a function's execution is partially complete. This is accomplished using `SBValue`'s reference to an `SBAddress`. From there, you can compare the addresses of all the Objective-C methods you've obtained to the starting address of the synthetic `SBSymbol`. If two addresses match, then you can swap out the stripped (synthetic) method name and replace it with the function name that was obtained with the Objective-C runtime.

Don't worry: You'll explore this systematically using LLDB's **script** command before you go building this Python script.

50 Shades of Ray

Included in the starter directory is an application called **50 Shades of Ray**. A well-chosen name (in my humble opinion) for a project that showcases the many faces of Ray Wenderlich. There's gentle Ray, there's superhero Ray, there's confused Ray, there's even goat BFF Ray!

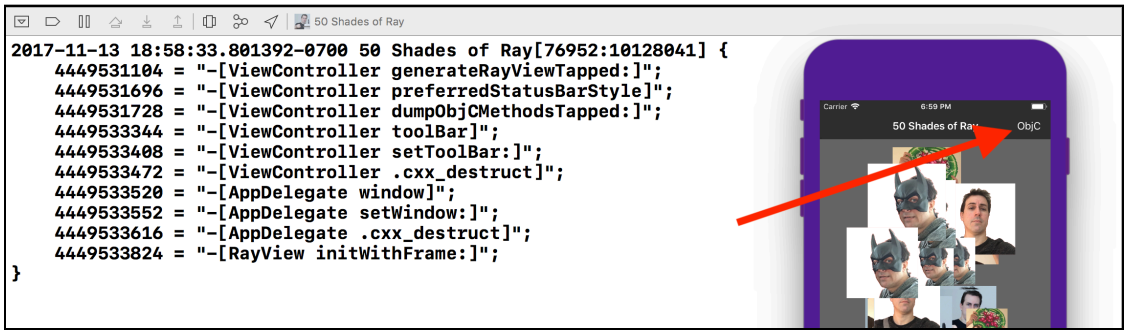


When tapping the `UIButton` at the bottom, a randomly generated picture of Ray pops up in a `UIView` of random size.

Wow, that will make *billions* on the App Store!

Open the **50 Shades of Ray** project and build and run the app. In the Xcode project, there are two schemes. Make sure you select the **50 Shades of Ray** scheme and not the **Stripped** scheme. You'll use that scheme later.

Once you've gotten your enjoyment out of generating random pictures of Ray, click on the **ObjC** `UIBarButtonItem` in the upper right hand corner.



This `UIBarButtonItem` is tied to an `IBAction` that will print out all the methods implemented by the main executable and displays them to stderr in your console. In fact, you can see the name of the method that triggered the console output within the console output!

Scan the console for the method **-[ViewController dumpObjCMethodsTapped:]**. It's this method which dumped all the Objective-C methods in the main executable.

Preceding the function is a number (in my case, `4449531728`), which holds the starting address for this Objective-C method.

Don't believe me? Pause execution and type the following into LLDB:

```
(lldb) image lookup -a 4449531728
```

You address will be different. This is hunting down the location of the address **4483016672** in memory and seeing where it relates in reference to your project.

```
Address: 50 Shades of Ray[0x00000001000017e0] (50 Shades of
Ray.__TEXT.__text + 624)
Summary: 50 Shades of Ray`-[ViewController dumpObjCMethodsTapped:] at
ViewController.m:36
```

Groovy. This is telling us the location in memory `4449531728` is what was loaded from **`-[ViewController dumpObjCMethodsTapped:]`**. Let's look at the code in this method.

Head on in to **ViewController.m** and hunt for the **dumpObjCMethodsTapped:**

The exact details don't need to be covered too closely, but it's worth pointing out the following:

- All the Objective-C classes implemented in the main executable are enumerated through **`objc_copyClassNamesForImage`**.
- For each class, there's logic to grab all the class and instance methods.
- In order to grab the class methods for a particular Objective-C Class, you must get the **meta class**. No, that term was not made up by some hipster developer in tight jeans, plaid shirt & beard. The meta class is the class responsible for the static methods of a particular class. For example, all methods that begin with + are implemented by the meta Class and not the Class.
- All the methods are aggregated into a `NSMutableDictionary`, where the key for each of these methods is the location in memory where the function resides.

Using script to guide your way

Time to use the **`script`** LLDB command to explore the lldb module APIs and build a quick POC to see how you're going to tackle finding the starting address of a function in memory.

In the LLDB console, set a breakpoint on **NSLog**:

```
(lldb) b NSLog
```

You'll get multiple `SBBreakpointLocation` hits. That's fine. Now continue running the application.

Tap on the **ObjC** `UIBarButtonItem` in the upper right corner of the Simulator.

Execution will stop right before content is spat out to `stderr`.

Using the global variable **`lldb.frame`**, dig into what APIs are available to you to grab the starting address of the `NSLog` function.

Start with the global variable and build from there.

```
(lldb) script print lldb.frame
```

You'll get the `__str__()` representation of the `SBFrame`. Nothing new.

```
frame #0: 0x000000010b472390 Foundation`NSLog
```

If you decided to use **`gdocumentation`** to search documentation for `SBFrame` (from Chapter 23, "Script Bridging Classes and Hierarchy," you'll see `SBFrame` has a few potential candidates for getting the start address of a function.

`pc` looks interesting to grab the `RIP` regster (x64) or the `PC` (ARM64), but that will only work at the start of a function. You need to grab the starting address from any offset inside the `SBFrame`.

Unfortunately, there are no APIs you can use in the `SBFrame` to get the starting address from any instruction offset within the function. You'll need to turn your attention to other classes referenced by the `SBFrame` to get what you need.

Grab the **`SBSymbol`** reference for the `SBFrame`:

```
(lldb) script print lldb.frame.symbol
```

The `SBSymbol` is responsible for the implementation offset address of `NSLog`. That is, the `SBSymbol` will tell you where this function is implemented in a module; it doesn't hold the actual address of where the `NSLog` was loaded into memory.

However, you can use the `SBAddress` property along with the **`GetLoadAddress`** API of `SBAddress` to find where the start location of `NSLog` is in your current process.

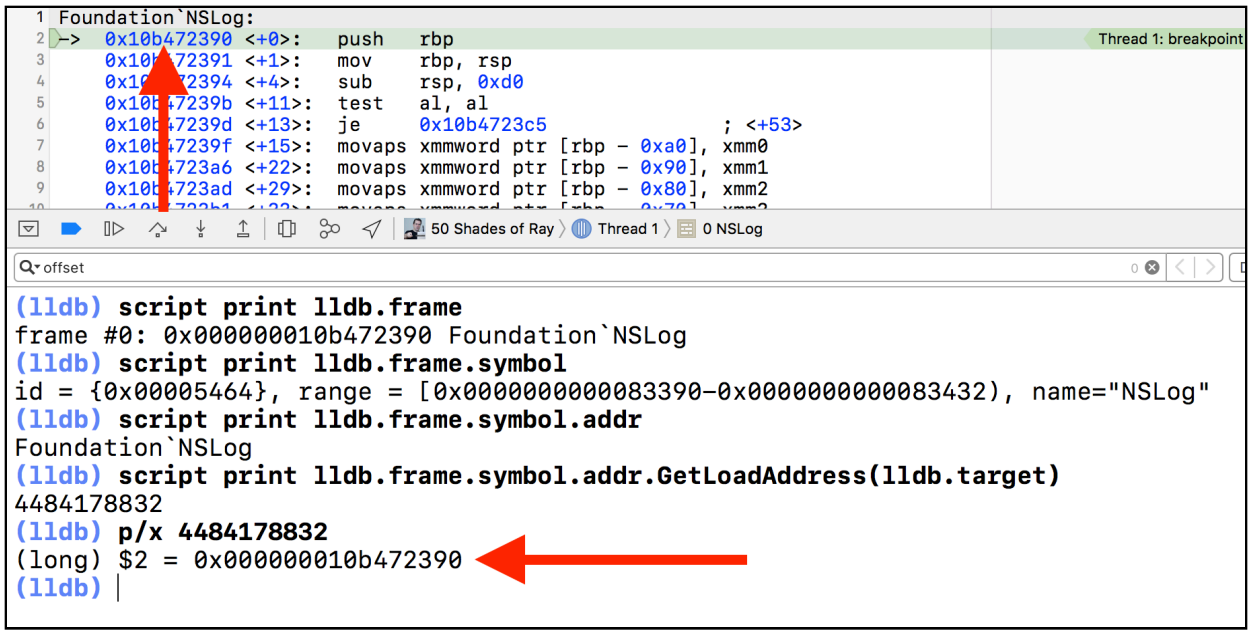
```
(lldb) script print lldb.frame.symbol.addr.GetLoadAddress(lldb.target)
```

You'll get a number in decimal. I got **4484178832**. Convert it to hex using LLDB and compare the output to the start address of `NSLog`:

```
(lldb) p/x 4484178832
```

I got **0x000000010b472390** as my hexadecimal representation.

Compare your output with the starting address of `NSLog` to see if they match.



Woot! A match! That's your path to resymbolication redemption.

lldb.value with NSDictionary

Since you're already here, you can explore one more thing. How are you going to parse this `NSDictionary` with all these addresses?

You'll copy the code, *almost* verbatim, that generates all the methods and apply it to an **`EvaluateExpression`** API to get an `SBValue`.

You should still be paused at the beginning of `NSLog`. Jump to the calling frame, **-[ViewController dumpObjCMethodsTapped:]**.

```
(lldb) f 1
```

This will get to the previous frame, `dumpObjCMethodsTapped:`. You now have access to all variables within this method, including the **`retdict`** that's responsible for dumping out all the methods implemented within the main executable.

Grab the `SBValue` interpretation of the **`retdict`** reference.

```
(lldb) script print lldb.frame.FindVariable('retdict')
```

This will print the `SBValue` for `retdict`:

```
(__NSDictionaryM *) retdict = 0x000060800024ce10 10 key/value pairs
```

Since this an `NSDictionary`, you actually want to dereference this value so you can enumerate it.

```
(lldb) script print lldb.frame.FindVariable('retdict').deref
```

You'll get some more relevant output (which is truncated):

```
(__NSDictionaryM) *retdict = {
  [0] = {
    key = 0x000060800002bb80 @"4411948768"
    value = 0x000060800024c660 @"-[AppDelegate window]"
  }
  [1] = {
    key = 0x000060800002c1e0 @"4411948592"
    value = 0x000060800024dd10 @"-[ViewController toolBar]"
  }
  [2] = {
    key = 0x000060800002bc00 @"4411948800"
    value = 0x000060800024c7e0 @"-[AppDelegate setWindow:]"
  }
  [3] = {
    key = 0x000060800002bba0 @"4411948864"
    value = 0x000060800004afe0 @"-[AppDelegate .cxx_destruct]"
  }
```

It's this you want to start with, since this prints out all the `values` for the `keys`.

Make a **`lldb.value`** out of this `SBValue` and assign it to a variable **`a`**.

```
(lldb) script a = lldb.value(lldb.frame.FindVariable('retdict').deref)
```

This is one of those times where I would prefer to work with an `lldb.value` over an `SBValue`. From here, you can easily explore the values within this `NSDictionary`.

Print the first value within this `lldb.value` `NSDictionary`.

```
(lldb) script print a[0]
```

From there, you can have either the `key` or `value` that you can print out.

Print out the `key` first:

```
(lldb) script print a[0].key
```

You'll get something similar to the following:

```
(__NSCFString *) key = 0x000060800002bb80 @"4411948768"
```

Print the `value`:

```
(lldb) script print a[0].value
```

This will print something similar to the following:

```
(__NSCFString *) value = 0x000060800024c660 @"-[AppDelegate window]"
```

```
(lldb) script print a[0]
(__lldb_autogen_nspair) [0] = {
  key = 0x000060800002bb80
  value = 0x000060800024c660
}
(lldb) script print a[0].key
(__NSCFString *) key = 0x000060800002bb80 @"4411948768"
(lldb) script print a[0].value
(__NSCFString *) value = 0x000060800024c660 @"-[AppDelegate window]"
(lldb)
```

If you only want the return value without the referencing address, you'll need to cast this `lldb.value` back into a `SBValue` then grab the `description`.

```
(lldb) script print a[0].value.sbvalue.description
```

This will get you the desired **`-[AppDelegate window]`** for output. Note you may have a different method.

If you wanted to dump all `keys` in this `lldb.value` `a` instance, you can use Python List comprehensions to dump all the keys out.

```
(lldb) script print '\n'.join([x.key.sbvalue.description for x in a])
```

You'll get output similar to the following:

```
4411948768
4411948592
4411948800
4411948864
4411948656
4411948720
4411949072
4411946944
4411946352
4411946976
```

Same approach for `values`:

```
(lldb) script print '\n'.join([x.value.sbvalue.description for x in a])
```

You now know how to parse this `NSDictionary` if, hypothetically, it were to be placed in some JIT code…

The plan is to copy the code from the `dumpObjCMethodsTapped:` into the Python script, and have it execute as JIT code. From there, you'll use the same procedure to parse it out from the `NSDictionary`.

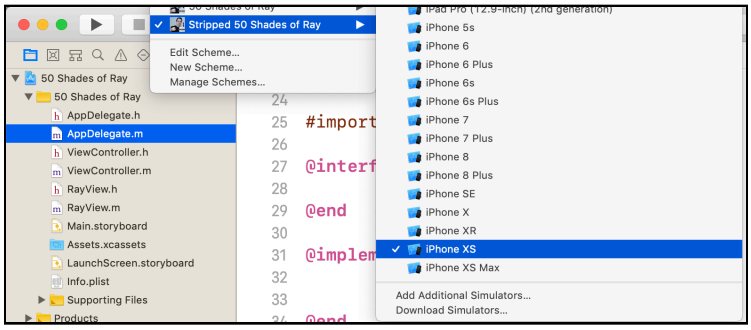
Sounds good? Get your gameplan ready and head on in to the next section!

The "stripped" 50 Shades of Ray

Yeah, that title got your attention, didn't it?

Within the Xcode schemes of the 50 Shades of Ray executable, there is a scheme named **Stripped 50 Shades of Ray**.

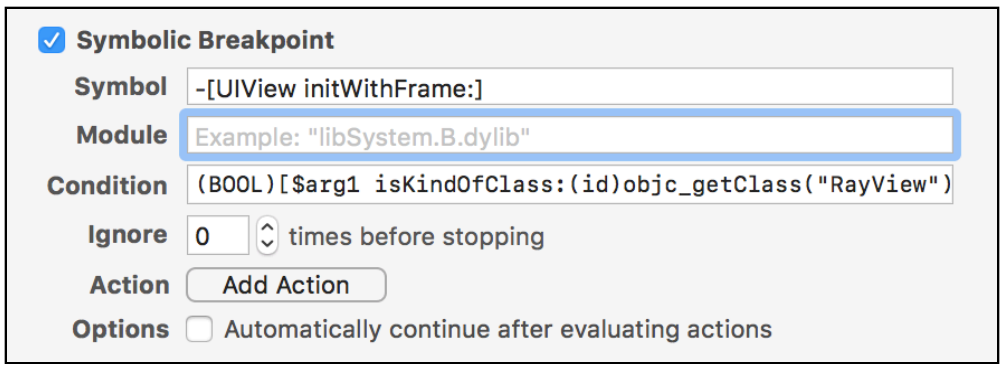
Stop the execution of the current process (⌘ + .) and select the **Stripped 50 Shades of Ray** Xcode scheme.



This scheme will build a debug executable, but remove the debugging information that you have become accustomed to in your day-to-day development cycles.

Build and run the executable. Included within this project is a **shared symbolic breakpoint**. Enable this breakpoint.

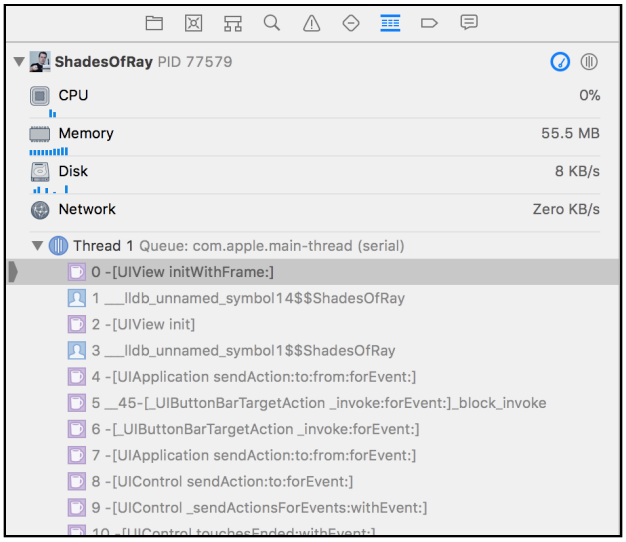
There's no need to modify this symbolic breakpoint, but it's worth noting what this breakpoint will do.



This breakpoint will stop on **`-[UIView initWithFrame:]`** and has a condition to only stop if the `UIView` is of type **`RayView`**, a subclass of `UIView`. This `RayView` is responsible for displaying the lovely images of Ray Wenderlich within the application.

Tap the **Generate a Ray!** button. Execution will stop on **`-[UIView initWithFrame:]`** method.

Take a look at the stack trace.



There's something interesting about stack frame 1 & 3: There's no debug information in there. LLDB has defaulted to generating a synthetic function name for those methods.

Confirm this in LLDB.

In LLDB, make sure you are in the starting frame (`initWithFrame:`):

```
(lldb) f 0
```

Use **script** to see if it's synthetic or not:

```
(lldb) script lldb.frame.symbol.synthetic
```

You'll get `False`. Makes sense, because you know this is `initWithFrame:`. Jump to one of the synthetic frames:

```
(lldb) f 1
```

Execute the previous `script` logic:

```
(lldb) script lldb.frame.symbol.synthetic
```

You'll get `True` this time.

This is enough research to get you going with the Python script.

Building sbt.py

Included within the **starter** folder is a Python script named **sbt.py**.

Stick this script into your **~/lldb** directory. Provided you've installed the **lldbinit.py** script, this will load all the Python files into the LLDB directory.

If you didn't follow along in Chapter 26, "SB Examples, Improved Lookup", you can manually install the `sbt.py` by modifying your **~/.lldbinit** file.

Once you've placed the sbt.py file into the **~/lldb directory**, reload your commands in **~/.lldbinit** using the **reload_script** you created in Chapter 23, "Script Bridging Classes and Hierarchy".

Check and see if LLDB correctly recognizes the `sbt` command:

```
(lldb) help sbt
```

You'll get some help text if LLDB recognizes the command. This will be the starting point for the `sbt` command.

Open this file up and jump down to **`generateExecutableMethodsScript`**. There's something interesting here that's worth pointing out.

Do you remember in the previous chapter, how I mentioned **lldb.value** is slooooooooooooooooow? If you're exploring a huge executable with lots of methods, the amount of time it takes for Python to go through every value in an `NSDictionary` takes forever.

Instead, you don't need to grab *every* reference to *every* single function in your `NSDictionary`. You only need to grab the locations of the start of each function in the stack trace.

```
def generateExecutableMethodsScript(frame_addresses):
    frame_addr_str = 'NSArray *ar = @['
    for f in frame_addresses:
        frame_addr_str += '@"' + str(f) + '",'

    frame_addr_str = frame_addr_str[:-1]
    frame_addr_str += '];'

    # ############################
```

```
    # Truncated content...
    # ############################

    command_script += frame_addr_str
    command_script += r'''
NSMutableDictionary *stackDict = [NSMutableDictionary dictionary];
[retdict keysOfEntriesPassingTest:^BOOL(id key, id obj, BOOL *stop) {
  if ([ar containsObject:key]) {
    [stackDict setObject:obj forKey:key];
    return YES;
  }
  return NO;
}];
stackDict;
'''
    return command_script
```

This is a pretty sweet optimization, because instead of evaluating potentially thousands (if not tens of thousands) of Objective-C methods, you'll only need to evaluate less than 20 keys or so in an `NSDictionary`, or whatever amount of synthetic functions are in the stack frame.

With the symbolic breakpoint still active and program stopped, give the script a run.

Just a normal stack frame will be printed out that doesn't have logic to resymbolicate the symbols.

It's time to make a few modifications to fix that.

Implementing the code

The JIT code is already set up. All you need to do is just call it, then compare the return `NSDictionary` against any synthetic `SBValues`.

Inside **`processStackTraceStringFromAddresses`**, search for the following comments:

```
    # New content start 1
    # New content end 1
```

Stick your new code here to call the JIT code to generate a list of potential methods in a `NSDictionary`:

```
    # New content start 1
    methods = target.EvaluateExpression(script, generateOptions())
    methodsVal = lldb.value(methods.deref)
    # New content end 1
```

You've called the code that returns the `NSDictionary` representation and assigned it to the `SBValue` instance variable **`methods`**.

You can cast the **SBValue** into a **lldb.value** (technically it's just a **value**, but you might get confused if I don't have the module in there) and assign it to the variable **methodsVal**.

Now for the final part of Python code. All you need to do is determine if a `SBFrame`'s `SBSymbol` is `synthetic` or not and perform the appropriate logic.

Search the following commented out code further down in `processStackTraceStringFromAddresses`:

```
    # New content start 2
    name = symbol.name
    # New content end 2
```

Change this to look like the following:

```
# New content start 2
if symbol.synthetic: # 1
    children = methodsVal.sbvalue.GetNumChildren() # 2
    name = symbol.name + r' ... unresolved womp womp' # 3

    loadAddr = symbol.addr.GetLoadAddress(target) # 4

    for i in range(children):
        key = long(methodsVal[i].key.sbvalue.description) # 5
        if key == loadAddr:
            name = methodsVal[i].value.sbvalue.description # 6
            break
else:
    name = symbol.name # 7

# New content end 2

offset_str = ''
```

Breaking this down, you have the following:

1. You're enumerating the frames, which occur outside the scope of this code block. For each symbol, a check is performed to see if the symbol is `synthetic` or not. If it is, the memory address will be compared to the `NSDictionary` of addresses that were gathered.

2. This will grab the number of children in the `lldb.value` that will be enumerated to see if there's a match from the Objective-C list of classes.

3. Either way, a valid reference to the `name` variable needs to be produced for the display of the stack trace. You're opting to say you know this is a synthetic function, but fail to resolve it if your upcoming logic fails to produce a result.

4. This gets the address in memory to the synthetic function in question.

5. The `key` value given by the `lldb.value` is internally made up from a `NSNumber`, so you need to grab the **`description`** of this method and cast it into a number. Confusingly, it's assigned to a Python variable named `key` as well.

6. If the `key` variable is equal to the `loadAddr`, then you have a match. Assign the `name` variable to the `description` of the variable in the `NSDictionary`.

That should be it. Save your work and reload your LLDB contents using `reload_script` and give it a go.

Provided you are still in the `Stripped 50 Shades of Ray` scheme and are paused in the symbolic breakpoint that stops only in `UIView`'s `initWithFrame:` (with the special condition), run the **`sbt`** command in the debugger to see if the originally unavailable frames 1 & 3 can be read.

```
(lldb) sbt bt
frame #0: 0x1053fe694 UIKit`-[UIView initWithFrame:]
frame #1: 0x103cf53ac ShadesOfRay`-[RayView initWithFrame:] + 924
frame #2: 0x1053fdda2 UIKit`-[UIView init] + 62
frame #3: 0x103cf45bf ShadesOfRay`-[ViewController
generateRayViewTapped:] + 79
```

Beautiful.

Where to go from here?

Congratulations! You've used the Objective-C runtime to successful resymbolicate a stripped binary! It's crazy what you can do with the proper application of Objective-C.

There are still a few holes in this script. This script doesn't play nice with Objective-C blocks. However, a careful study of how blocks are implemented as well as exploring the lldb Python module *might* reveal a way to indicate Objective-C block functions that have been stripped away.

In addition, this script will not work with an iOS executable in release mode. LLDB will not find the functions for a synthetic `SBSymbol` to reference the start address. This means that you would have to manually search upwards in the ARM64 assembly until you stumbled across an assembly instruction that looked like the start of a function (can you guess which instruction(s) to look for?).

If those script extensions don't interest you, try your luck with figuring out how to resymbolicate a Swift executable. The challenge definitely goes up by an order of magnitude, but it's still within the realm of possibility to do with LLDB. Have fun!

Chapter 28: SB Examples, Malloc Logging

For the final chapter in this section, you'll go through the same steps I myself took to understand how the **MallocStackLogging** environment variable is used to get the stack trace when an object is created.

From there, you'll create a custom LLDB command which gives you the stack trace of when an object was allocated or deallocated in memory — even after the stack trace is long gone from the debugger.

Knowing the stack trace of where an object was created in your program is not only useful for reverse engineering, but also has great use cases in your typical day-to-day debugging. When a process crashes, it's incredibly helpful to know the history of that memory and any allocation or deallocation events that occurred before your process went off the deep end.

This is another example of a script using stack-related logic, but this chapter will focus on the complete cycle of how to explore, learn, then implement a rather powerful custom command.

Setting up the scripts

You have a couple of scripts to use (and implement!) for this chapter. Let's go through each one of them and how you'll use them:

- **msl.py**: This is the command (which is an abbreviation for `MallocStackLogging`) is the script you'll be working on in this chapter. This has a basic skeleton of the logic.

- **lookup.py**: Wait — you already made this command, right? Yes, but I'll give you my own version of the `lookup` command that adds a couple of additional options at the price of uglier code. You'll use one of the options to filter your searches to specific modules within a process.
- **sbt.py**: This command will take a backtrace with unsymbolicated symbols, and symbolicate it. You made this in the previous chapter, and you'll need it at the very end of this chapter. And in case you didn't work through the previous chapter, it's included in this chapter's resources for you to install.
- **search.py**: This command will enumerate all objects in the heap and search for a particular subclass. This is a *very* convenient command for quickly grabbing references to instances of a particular class.

> **Note**: These scripts come from https://github.com/DerekSelander/lldb. If I need a tool that I don't have, I'll build it, and stick it in the above repo. Check it out for some other novel ideas for LLDB scripts. It's important to note that a lot of scripts in the above repo have dependencies on other files included in the repo, so if you only download one script, it might not compile until the full set of files is included.

Now for the usual setup. Take all the Python files found in the **starter** directory for this chapter and copy them into your `~/lldb` directory. I am assuming you have the `lldbinit.py` file already set up, found in Chapter 26, "SB Examples, Improved Lookup."

Launch an LLDB session in Terminal and go through all the `help` commands to make sure each script has loaded successfully:

```
(lldb) help msl
(lldb) help lookup
(lldb) help sbt
(lldb) help search
```

MallocStackLogging explained

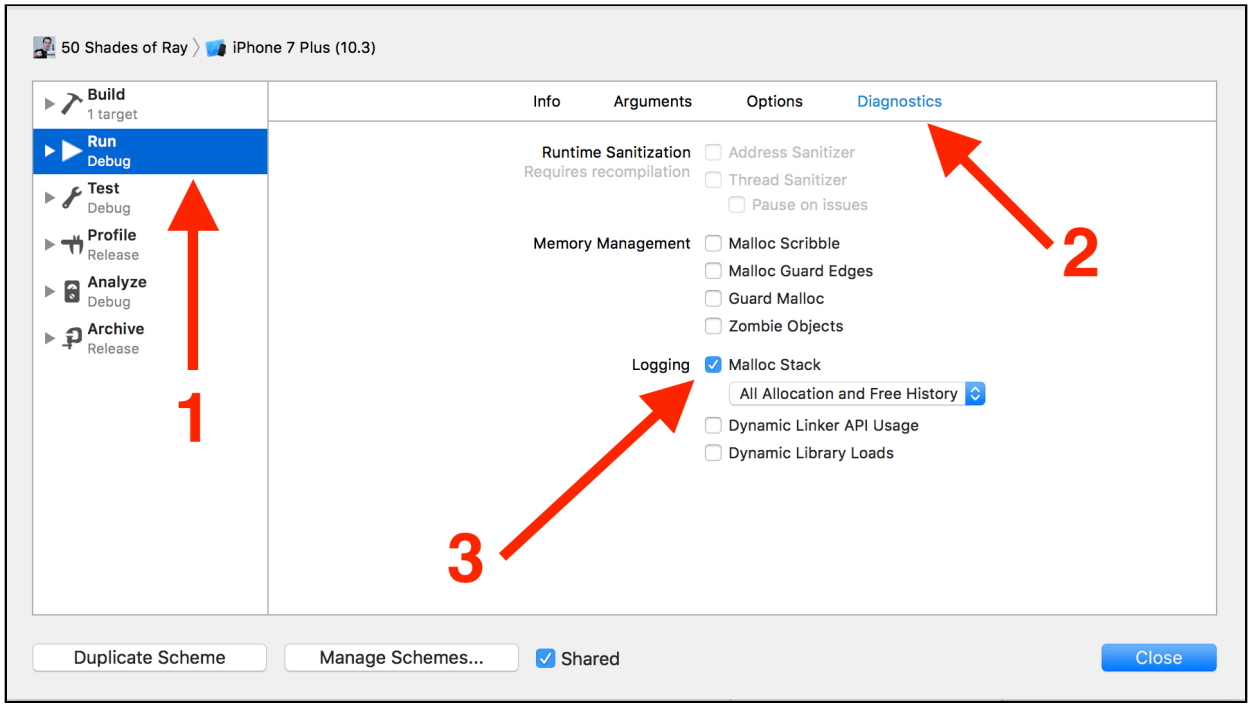
In case you're unfamiliar with the `MallocStackLogging` environment variable, I'll describe it and show how it's typically used.

When the `MallocStackLogging` environment variable is passed into a process, and is set to `true`, it'll monitor allocations and deallocations of memory on the heap. Pretty neat!

Included within the **starter** directory is the **50 Shades of Ray** Xcode project with some additional logic for this chapter. Open the project.

Before you run it, you'll need to modify the scheme for your purposes. Select the **50 Shades of Ray** scheme (make sure there's no "Stripped" in the name), then press ⌘ + **Shift** + < to edit the scheme.

Select **Run**, then **Diagnostics**, then select **Malloc Stack**, then **All Allocation and Free History**.



Once you've enabled this environment variable, build the **50 Shades of Ray** program and run it on the **iPhone 8 Simulator**.

If the `MallocStackLogging` environment variable is enabled, you'll see some output from the LLDB console similar to the following:

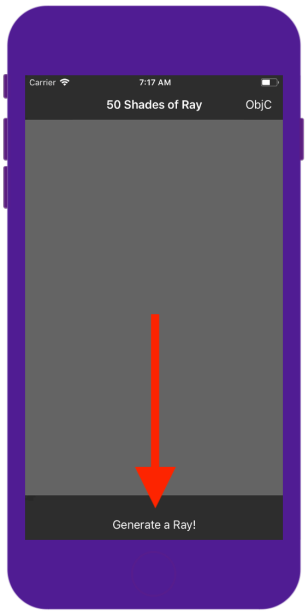
```
ShadesOfRay(12911,0x104e663c0) malloc: stack logs being written into /
tmp/stack-logs.12911.10d42a000.ShadesOfRay.gjehFY.index

ShadesOfRay(12911,0x104e663c0) malloc: recording malloc and VM allocation
stacks to disk using standard recorder

ShadesOfRay(12911,0x104e663c0) malloc: process 12673 no longer exists,
stack logs deleted from /tmp/stack-logs.
12673.11b51d000.ShadesOfRay.GVo3li.index
```

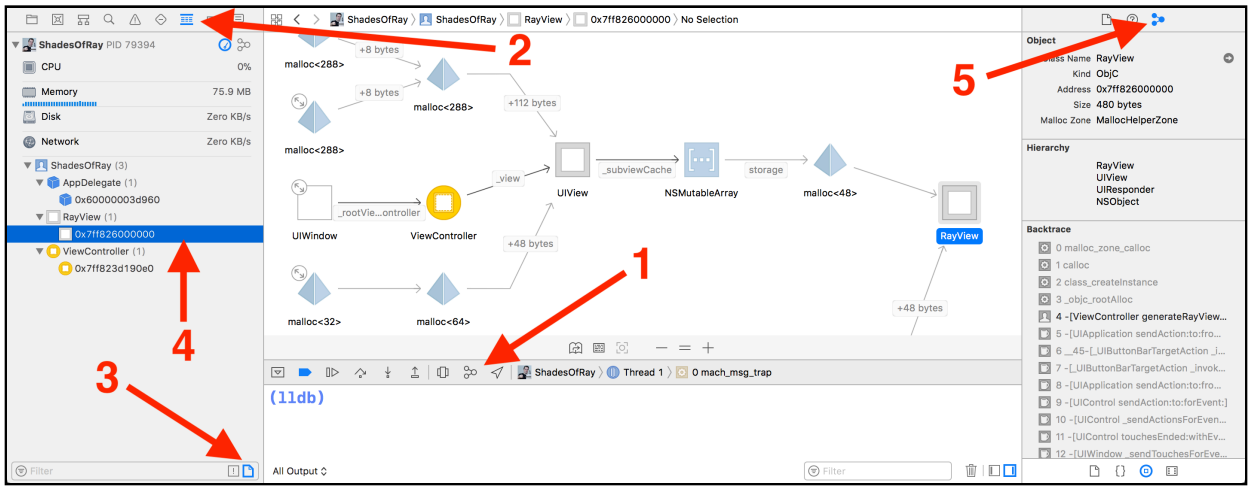
Don't worry about the details of the output; simply look for the presence of output like this as it indicates the `MallocStackLogging` is working properly.

While the app is running, click the **Generate a Ray** button at the bottom.



Once a new Ray is created (that is, you see an instance of Ray Wenderlich's amazingly innovative & handsome face pop up in the Simulator), perform the following steps:

1. Select the **Debug Memory Graph** located at the top of the LLDB console in Xcode.
2. Select the **Show the Debug navigator** in the left panel.
3. At the bottom of the left panel select the **Show only content from workspace**.
4. Select the reference to the **RayView**.
5. In the right panel of Xcode, make sure the **Show the Memory Inspector** is selected.



Once you've jumped through all those hoops, you'll have the exact stack trace of where this `RayView` instance was created through the **Backtrace** section in Xcode. How cool is that?! The authors of Xcode (and its *many* modules) have made our lives a bit easier with these memory debugging features!

Plan of attack

You know it's possible to grab a stack trace for an instantiated object, but you're going to do one better than Apple.

Your command will be able to turn on the `MallocStackLogging` functionality at will through LLDB, which means you won't have to rely on an environment variable. This has the additional benefit that you won't need to restart your process in case you forget to turn it on during a debug session.

So how are you going to figure out how this `MallocStackLogging` feature works?

When I am absolutely clueless as to where to begin when exploring built-in code, I follow the rather loose process below and alter queries, depending on the scenario or the output:

- I look for chokepoints where I can safely assume some logic of interest will be executed in a process I am attached to. If I know I can replicate something of interest, I'll force that action to occur while monitoring it.
- When monitoring the code of interest, I'll use various tools like LLDB or DTrace (which you'll learn about in the next chapter) to find the module which holds the code of interest. Again, the module is a dynamic library, framework, `NSBundle`, or something of that sort.
- Once I find the module of interest, I'll dump all the code within the module, then filter for what I need using various custom scripts like **lookup.py**.
- If I find a particular function that looks relevant to my interests, I'll first try Googling it. I'll often find some incredibly useful hints on https://opensource.apple.com/ that reveals how I can use what I've found.
- Searching through Apple's opensource URLs, I'll grab as much context as I can about the code of interest. Sometimes there's code in the C/C++ source file that will give me an idea of how to formulate the parameters into the function, or perhaps I'll get a description of the code or its purpose in the header file.

- If there's no documentation to be gained from Googling, I'll set breakpoints on the code of interest and see if I can trigger that function naturally. Once hit, I'll explore both the stack frames and registers to see what kind of parameters are being passed in, as well as the context it's used in.

You're going to follow the exact same steps to see where the code for `MallocStackLogging` resides, explore the module responsible for handling stack tracing logic, then explore any interesting code of interest within that module.

Let's get cracking!

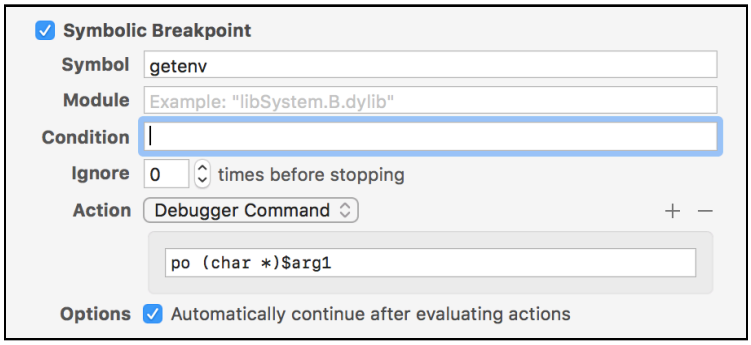
Hunting in getenv

`MallocStackLogging` is an environment variable passed into the process. This means the C `getenv` function is likely used to check if this argument is supplied, and perform additional logic if it is.

You need to dump all the items queried with `getenv` when the process starts up. You'll perform the same action you did in Chapter 16, "Hooking & Executing Code with dlopen & dlsym" by creating a symbolic breakpoint to dump the `char*` parameter when `getenv` is being called.

In Xcode, create a symbolic breakpoint with the following logic:

- **Symbol**: `getenv`
- **Action**: `po (char *)$arg1`
- **Automatically continue after evaluating actions**: yep!



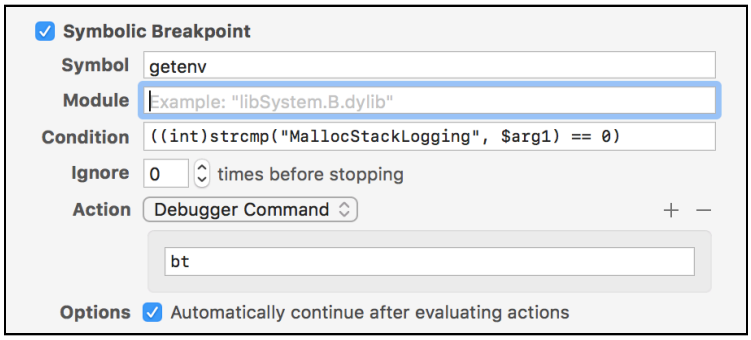
Build and run the program with the `MallocStackLogging` variable still checked.

From the output, you can see that somewhere in the startup process, there's code that checks for the presence of `MallocStackLogging`.



Modify your symbolic breakpoint to only dump the stack trace when the program is checking for the `MallocStackLogging` environment variable:

- **Symbol**: `getenv`
- **Condition**: `((int)strcmp("MallocStackLogging", $arg1) == 0)`
- **Action**: `bt`
- **Automatically continue after evaluating actions**: *¡Sí!*



Once your augmented symbolic breakpoint is set up, rerun the app.

You'll get a couple of stack traces in the console. Check out the very first one:

```
   * frame #0: 0x0000000112b4da26 libsystem_c.dylib`getenv
     frame #1: 0x0000000112c7dd53
libsystem_malloc.dylib`_malloc_initialize + 466
     frame #2: 0x0000000112ddcac1 libsystem_platform.dylib`_os_once + 36
     frame #3: 0x0000000112c7d849
libsystem_malloc.dylib`default_zone_malloc + 77
     frame #4: 0x0000000112c7d259
libsystem_malloc.dylib`malloc_zone_malloc + 103
     frame #5: 0x0000000112c7f44a libsystem_malloc.dylib`malloc + 24
```

```
    frame #6: 0x0000000112aa2947 libdyld.dylib`tlv_load_notification +
286
    frame #7: 0x000000010e0f68a9 dyld_sim`dyld::registerAddCallback(void
(*)(mach_header const*, long)) + 134
    frame #8: 0x0000000112aa1a0d
libdyld.dylib`_dyld_register_func_for_add_image + 61
    frame #9: 0x0000000112aa1be7 libdyld.dylib`_dyld_initializer + 47
```

Interesting... Check out stack frame 1:

```
frame #1: 0x0000000112c7dd53 libsystem_malloc.dylib`_malloc_initialize +
466
```

If I were an Apple author, I would likely be checking for an environment variable to conditionally see if my code should run right when it's initialized. This looks like it's doing the same, plus the module name, **libsystem_malloc.dylib** fits the bill for something implementing malloc stack logging related logic. Is this it? Maybe. Worth checking out? Totally!

Take a deeper dive into this module and see what it has to offer you.

Using the fancy, new & improved `lookup` command, explore all the methods implemented by the **libsystem_malloc.dylib** module that you can execute within your process.

Pause the app in the debugger, and then type the following in your LLDB console:

```
(lldb) lookup . -m libsystem_malloc.dylib
```

In iOS 12.0, I get 420 hits. I *could* gloss through all these methods, but I am getting increasingly lazy as a debugger person. Let's just hunt for everything that pertains to the word "log" (for logging) and see what we get. Type the following in LLDB:

```
(lldb) lookup (?i)log -m libsystem_malloc.dylib
```

I get 54 hits from using a case insensitive search for the word `log` inside the `libsystem_malloc.dylib` module.

This hit count is bearable enough to weed through.

Do any of those functions look interesting? Hell yeah! Here are some of the following functions that look interesting to me:

```
create_log_file

open_log_file_from_directory

__mach_stack_logging_get_frames
```

```
turn_off_stack_logging

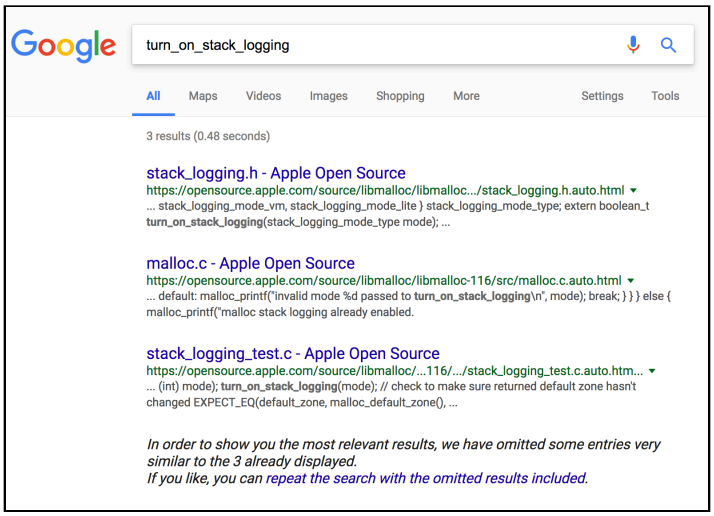
turn_on_stack_logging
```

Of my top 5, the **turn_on_stack_logging** and the **__mach_stack_logging_get_frames** look like they're worth checking out.

You've found the module of interest, as well as some functions worth further exploration. Time to jump over to Google and see what's out there.

Googling JIT function candidates

Google for any code pertaining to **turn_on_stack_logging**. Take a look at this search query:



At the time I wrote this, I got three hits from Google (well, it was actually eight hits with "exclude similar searches" off, but that's not the point).

These functions are not well-known and are not typically discussed in any circle outside of Apple. In fact, I am rather confident the majority of iOS application developers in Apple don't know about them either, because when would they use them for writing apps?

This stuff belongs to the low-level C developers of Apple, whom we totally take for granted.

From the Google search, check out the following code from the header file found at https://opensource.apple.com/source/libmalloc/libmalloc-116/private/stack_logging.h.auto.html:

```
typedef enum {
  stack_logging_mode_none = 0,
  stack_logging_mode_all,
  stack_logging_mode_malloc,
  stack_logging_mode_vm,
  stack_logging_mode_lite
} stack_logging_mode_type;

extern boolean_t turn_on_stack_logging(stack_logging_mode_type mode);
```

This is some **really good** information to work with. The `turn_on_stack_logging` function expects one parameter of type `int` (C enum). The enum **stack_logging_mode_type** tells you if you want the **stack_logging_mode_all** option, it will be at value 1.

You'll run an experiment by turning off the stack logging environment variable, execute the above function via LLDB, and see if Xcode is recording stack traces for any malloc'd object after you've called `turn_on_stack_logging`.

Before you do that, you'll first explore the other function, `__mach_stack_logging_get_frames`.

Exploring __mach_stack_logging_get_frames

Fortunately, for your exploration efforts, `__mach_stack_logging_get_frames` can also be found in the same header file. This function signature looks like the following:

```
extern kern_return_t __mach_stack_logging_get_frames(
                                        task_t task,
                            mach_vm_address_t address,
              mach_vm_address_t *stack_frames_buffer,
                            uint32_t max_stack_frames,
                                     uint32_t *count);
    /* Gets the last allocation record (malloc, realloc, or free) about
address */
```

This is a good starting point, but what if there are parameters you're not 100% sure how to obtain? For example, what's `task_t task` all about? This is basically a parameter which specifies the process you want this function to act on. But what if you didn't know that?

Using Google and searching for any implementation files that contain `__mach_stack_logging_get_frames` can be a big help when you're uncertain about things like this.

After a casual Googling, the https://llvm.org/svn/llvm-project/lldb/trunk/examples/darwin/heap_find/heap/heap_find.cpp URL provides insight to the first parameter that's expected within this function.

This file contains the following code:

```
  task_t task = mach_task_self();
  /* Omitted code.... */
      stack_entry->address = addr;
      stack_entry->type_flags = stack_logging_type_alloc;
      stack_entry->argument = 0;
      stack_entry->num_frames = 0;
      stack_entry->frames[0] = 0;

      err = __mach_stack_logging_get_frames(task,
                          (mach_vm_address_t)addr,
                              stack_entry->frames,
                                      MAX_FRAMES,
                         &stack_entry->num_frames);

      if (err == 0 && stack_entry->num_frames > 0) {
        // Terminate the frames with zero if there is room
        if (stack_entry->num_frames < MAX_FRAMES)
          stack_entry->frames[stack_entry->num_frames] = 0;
      } else {
        g_malloc_stack_history.clear();
      }
    }
  }
```

The `task_t` parameter has a easy way to get the task representing the current process through the `mach_task_self` function located in **libsystem_kernel.dylib**. You can confirm this yourself with the `lookup` LLDB command.

Testing the functions

To prevent you from getting bored to tears, I've already implemented the logic for the `__mach_stack_logging_get_frames` inside the app.

Hopefully, you still have the application running. If not, get the app running with `MallocStackLogging` still enabled.

It's always a good idea to build your proof-of-concept JIT code in Xcode first, and once it's working, then (and only then!) transfer it to your LLDB script. You're gonna hate your life if you try to write your POC JIT script code straight in LLDB first. Trust me.

In Xcode, navigate to the **stack_logger.cpp** file. `__mach_stack_logging_get_frames` was written in C++, so you'll need to use C++ code to execute it.

The only function in this file is `trace_address`:

```
void trace_address(mach_vm_address_t addr) {

  typedef struct LLDBStackAddress {
    mach_vm_address_t *addresses;
    uint32_t count = 0;
  } LLDBStackAddress;   // 1

  LLDBStackAddress stackaddress; // 2
  __unused mach_vm_address_t address = (mach_vm_address_t)addr;
  __unused task_t task = mach_task_self_;  // 3

  stackaddress.addresses = (mach_vm_address_t *)calloc(100,
                                sizeof(mach_vm_address_t)); // 4

  __mach_stack_logging_get_frames(task,
                                  address,
                stackaddress.addresses,
                                  100,
                  &stackaddress.count); // 5

  // 6
  for (int i = 0; i < stackaddress.count; i++) {

    printf("[%d] %llu\n", i, stackaddress.addresses[i]);
  }

  free(stackaddress.addresses); // 7
}
```

Breakdown time!

1. As you know, LLDB only lets you return one object to be evaluated. But, as a creative string-theory version of yourself, can create C structs that contain any types you want to be returned.

2. Declare an instance of said struct for use within the function.

3. Remember `mach_task_self` that was referenced earlier? The global variable `mach_task_self_` is the value returned when calling `mach_task_self`.

4. Since you're in a lower level, you don't have ARC to help you allocate items on the heap. You're allocating 100 `mach_vm_address_t`'s, which is more than enough to handle any stack trace.

5. The `__mach_stack_logging_get_frames` then executes. The `addresses` array of the `LLDBStackAddress` struct will be populated with the addresses if there's any stack trace information available.

6. Print out all the addresses that were found

7. Finally, the `mach_vm_address_t` objects you created are freed.

Time to give it a whirl!

LLDB testing

Make sure the app is running, then tap the **Generate a Ray!** button. Pause execution and enter the following into LLDB:

```
(lldb) search RayView -b
```

The search script will enumerate all objects of a certain type in the heap. This command will hunt for all `RayView` instances that are currently alive.

The `-b` option will give you the `--brief` functionality, free of the class's `description` or `debugDescription` method. Depending on the amount of Ray Wenderlich faces on your Simulator, you'll get a variable amount of hits.

I have three wondrously magical Ray Wenderlich faces on my simulator, so I get the following output:

```
(lldb) search RayView -b
RayView * [0x00007fa838414330]
RayView * [0x00007fa8384125f0]
RayView * [0x00007fa83860c000]
```

Grab any one of those addresses and execute the logic in the `trace_address` function:

```
(lldb) po trace_address(0x00007fa838414330)
```

You'll get output that looks like the following truncated snippet:

```
[0] 4533269637
[1] 4460190625
[2] 4460232164
[3] 4454012240
[4] 4478307618
[5] 4482741703
[6] 4478307618
[7] 4479898204
[8] 4479898999
[9] 4479899371
...
```

These are the actual addresses of the code where this object is created. Verify the first address is code in memory using `image lookup`:

```
(lldb) image lookup -a 4533269637
```

You'll get the details about that function:

```
Address: libsystem_malloc.dylib[0x000000000000f485]
(libsystem_malloc.dylib.__TEXT.__text + 56217)
Summary: libsystem_malloc.dylib`calloc + 30
```

There's more than one way to skin a memory address. Copy the address at frame three and use `SBAddress` to get the information out of this address:

```
(lldb) script print lldb.SBAddress(4454012240, lldb.target)
```

You'll get stack frame 3, like so:

```
ShadesOfRay`-[ViewController generateRayViewTapped:] + 64 at
ViewController.m:38
```

Navigating a C array with lldb.value

You'll again use the **lldb.value** class to parse the return value of this C struct which was generated inline while executing this function.

Set a GUI breakpoint at the end of the `trace_address` function.

```
11
12 void trace_address(mach_vm_address_t addr) {
13
14   typedef struct LLDBStackAddress {
15     mach_vm_address_t *addresses;
16     uint32_t count = 0;
17   } LLDBStackAddress;
18
19   LLDBStackAddress stackaddress;
20   mach_vm_address_t address = (mach_vm_address_t)addr;
21   task_t task = mach_task_self_;
22   stackaddress.addresses = (mach_vm_address_t *)calloc(100, sizeof(mach_vm_address_t));
23   __mach_stack_logging_get_frames(task, address, stackaddress.addresses, 100, &stackaddress.count);
24
25   for (int i = 0; i < stackaddress.count; i++) {
26
27     printf("[%d] %llu\n", i, stackaddress.addresses[i]);
28   }
29
30   free(stackaddress.addresses);
31 }
32
```

Use LLDB to execute the same function, but honor breakpoints, and remember to replace the address with one of your `RayView` instances:

```
(lldb) e -lobjc++ -O -i0 -- trace_address(0x00007fa838414330)
```

Execution will stop on the final line of `trace_address`. You know the drill. Grab the reference to the C struct `LLDBStackAddress`, `stackaddress`.

```
(lldb) script print lldb.frame.FindVariable('stackaddress')
```

If successful, you'll get the synthetic format of the `stackaddress` variable:

```
(LLDBStackAddress) stackaddress = {
  addresses = 0x00007fa838515cd0
  count = 25
}
```

Cast this struct into a `lldb.value` and call the reference **`a`**:

```
(lldb) script a = lldb.value(lldb.frame.FindVariable('stackaddress'))
```

Ensure `a` is valid:

```
(lldb) script print a
```

You can now easily reference the variables you declared in the `LLDBStackAddress` struct inside the `lldb.value`. Type the following into LLDB:

```
(lldb) script print a.count
```

You'll get the stack frame count:

```
(uint32_t) count = 25
```

What about the `addresses` array inside the `LLDBStackAddress` struct?

```
(lldb) script print a.addresses[0]
```

That's the memory address of the first frame. What about that `generateRayViewTapped:` method found in frame 3?

```
(lldb) script print a.addresses[3]
```

You'll get something similar to:

```
(mach_vm_address_t) [3] = 4454012240
```

Do you see how this tool is coming together? From finding chokepoints of items of interest, to exploring code in modules, to researching tidbits of useful information in https://opensource.apple.com/, to implementing proof of concepts in Xcode before jumping to LLDB Python code, there's a lot of power under the hood.

Don't slow down — it's *command implementin'* time!

Turning numbers into stack frames

Included within the **starter** directory for this chapter is the **msl.py** script for malloc script logging. You've already installed this `msl.py` script earlier in the "Setting up the scripts" section.

Unfortunately, this script doesn't do much at the moment, as it doesn't produce any output. Time to change that.

Open up `~/lldb/msl.py` in your favorite editor. Find **`handle_command`** and add the following code to it:

```
command_args = shlex.split(command)
parser = generateOptionParser()
try:
    (options, args) = parser.parse_args(command_args)
except:
    result.SetError(parser.usage)
    return

cleanCommand = args[0]
process = debugger.GetSelectedTarget().GetProcess()
frame = process.GetSelectedThread().GetSelectedFrame()
target = debugger.GetSelectedTarget()
```

All this logic shouldn't be new to you, as it's the "preamble" required to start up the command. The only thing of interest is you opted to omit the `posix=False` argument that's sometimes used in the `shlex.split(command)`. There's no need to provide this parameter, since this command won't be handling any weird backslash or dash characters. This means the parsing of the output from the `options` and `args` variables is much cleaner as well.

Now that you have the basic script going, implement the meat of this script right below the code you just wrote:

```
# 1
script = generateScript(cleanCommand, options)

# 2
sbval = frame.EvaluateExpression(script, generateOptions())

# 3
if sbval.error.fail:
    result.AppendMessage(str(sbval.error))
    return

val = lldb.value(sbval)
addresses = []

# 4
```

```
for i in range(val.count.sbvalue.unsigned):
    address = val.addresses[i].sbvalue.unsigned
    sbaddr = target.ResolveLoadAddress(address)
    loadAddr = sbaddr.GetLoadAddress(target)
    addresses.append(loadAddr)

# 5
retString = processStackTraceStringFromAddresses(
                                        addresses,
                                           target)

# 6
freeExpr = 'free('+str(val.addresses.sbvalue.unsigned)+')'
frame.EvaluateExpression(freeExpr, generateOptions())
result.AppendMessage(retString)
```

Here are the items of interest:

1. Use the `generateScript` function I supplied, which returns a string containing roughly the same code as in the `trace_address` function.
2. Execute the code. You know this will return an `SBValue`.
3. Do a sanity check to see if the `EvaluateExpression` fails. If it does, dump out the error and exit early.
4. This for-loop will enumerate the memory addresses in the `val` object, which are the output of the `script` code, and pull them out into the `addresses` list.
5. Now that the addresses are pulled out into a list, you pass that list to a predefined function for processing. This will return the stack trace string you'll spit out.
6. Finally, you manually allocate memory, as you're a good memory citizen and always clean up after yourself. Most of these scripts you've written leak memory, but now that you're getting more advanced with this stuff, it's time to do the right thing and `free` any allocated memory.

Jump back to the Xcode LLDB console and reload your stuff:

```
(lldb) reload_script
```

Provided you have no errors, grab a reference to a `RayView` using the `search` LLDB command:

```
(lldb) search RayView -b
```

Just for kicks, here's another way to search for all `UIViews` whose class is implemented in the `ShadesOfRay` module:

```
(lldb) search UIView -m ShadesOfRay -b
```

Once you have a reference to a `RayView`, run your newly created `msl` command on it, like so:

```
(lldb) msl 0x00007fa838414330
```

You'll get your expected output just like in Xcode!

```
frame #0 : 0x11197d485 libsystem_malloc.dylib`calloc + 30
frame #1 : 0x10d3cbba1 libobjc.A.dylib`class_createInstance + 85
frame #2 : 0x10d3d5de4 libobjc.A.dylib`_objc_rootAlloc + 42
frame #3 : 0x10cde7550 ShadesOfRay`-[ViewController
generateRayViewTapped:] + 64
frame #4 : 0x10e512d22 UIKit`-[UIApplication
sendAction:to:from:forEvent:] + 83
```

Congratulations! You've created a script that will give you the stack trace for an object. Now it's time to level up and give this script some cool options!

Stack trace from a Swift object

OK — I know you want me to talk about Swift code. You'll cover a Swift example as well.

Included in the 50 Shades of Ray app is a Swift module, ingeniously named **SomeSwiftModule**. Within this module is a class named `SomeSwiftCode` with a static variable to get your singleton quota going.

The code in **SomeSwiftCode.swift** is about as simple as you can get:

```
public final class SomeSwiftCode {
  private init() {}
  static let shared = SomeSwiftCode()
}
```

You'll use LLDB to call this singleton and examine the stack trace where this function was created.

First off, you have to import your Swift modules! Enter the following into LLDB:

```
(lldb) e -lswift -O -- import SomeSwiftModule
```

You'll get no result if the above was successful.

In LLDB, access the singleton, like so:

```
(lldb) e -lswift -O -- SomeSwiftCode.shared
```

You'll get the address to this object:

```
<SomeSwiftCode: 0x600000033640>
```

Now you'll pass this address in to the `msl` command. But simply copying and pasting from the current output is waaaaaaaaaaaaaaay too easy. Use the `search` command instead and search for all subclasses of `SwiftObject`:

```
(lldb) search SwiftObject
```

You'll get something like the following:

```
<__NSArrayM 0x6000004578b0>(
SomeSwiftModule.SomeSwiftCode
)
```

Again, Swift tries to hide the pointer from you in `description`. That's part of its magic!

Use the `--brief` (`-b`) option one final time in the `search` command to grab the instance and ignore the object's `description` method.

```
(lldb) search SwiftObject -b
```

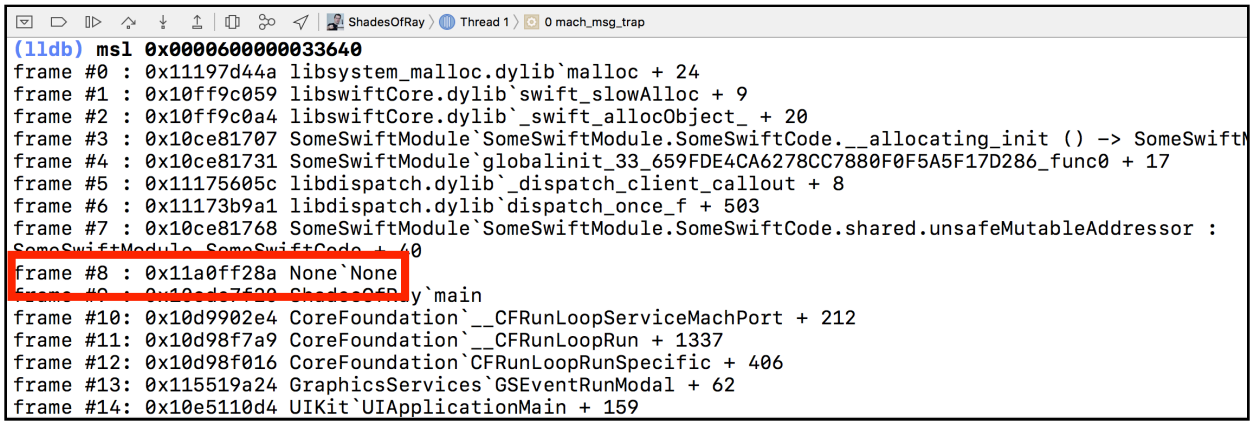
This will grab the mangled name, but it's the same reference in memory!

```
_TtC15SomeSwiftModule13SomeSwiftCode * [0x0000600000033640]
```

Use the `msl` command on this address:

```
(lldb) msl 0x0000600000033640
```

You'll get your expected stack trace.



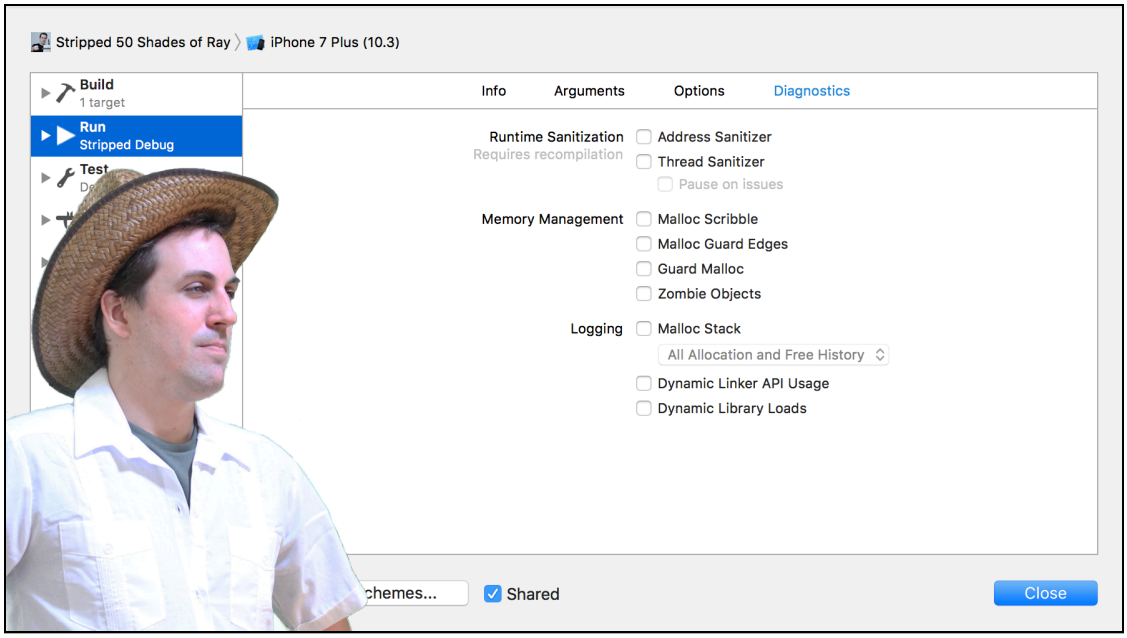
The highlighted frame here is clearly the frame where you call the singleton accessor from LLDB. Yours might be different.

Let's jump to one final topic I want to discuss briefly: how to build these scripts so you "Don't Repeat Yourself" when creating functionality in your LLDB scripts.

DRY Python code

Stop the app! In the schemes, select the **Stripped 50 Shades of Ray** Xcode scheme.

Ensure the `MallocStackLogging` environment variable is unchecked in the `Stripped 50 Shades of Ray` scheme.



Good. Ray approves.

Time to try out the `turn_on_stack_logging` function. Build and run the application.

As you found out in the previous chapter, the "Stripped 50 Shades of Ray" scheme strips the main executable's contents so there's no debugging information available. Remember that factoid when you use the `msl` command.

Once the application is up and running, tap the **Generate a Ray!** button to create a new instance of the `RayView`. Since the `MallocStackLogging` isn't enabled, let's see what happens...

Pause execution and search for all `RayViews` by typing the following into LLDB:

```
(lldb) search RayView -b
```

You'll get something like:

```
RayView * [0x00007fc23eb00620]
```

See if the `msl` command works on this address:

```
(lldb) msl 0x00007fc23eb00620
```

Nothing. That makes sense though, because the environment variable was not supplied to the process. Time to circle back and call `turn_on_stack_logging` to see what it does. Type the following in LLDB:

```
(lldb) po turn_on_stack_logging(1)
```

You'll get some output similar to the kind you get when you supply your process with the `MallocStackLogging` environment variable:

```
(lldb) search RayView -b
RayView * [0x00007f8250e030c0]
(lldb) msl 0x00007f8250e030c0

(lldb) po turn_on_stack_logging(1)
ShadesOfRay(24537,0x1147d43c0) malloc: stack logs being written into /tmp/stack-logs.
24537.11ea53000.ShadesOfRay.g0Y5EW.index
ShadesOfRay(24537,0x1147d43c0) malloc: recording malloc and VM allocation stacks to disk using
standard recorder
0x0000000000000001

(lldb)
```

Resume execution and create another instance of `RayView` by tapping the bottom button.

Once you've done that, pause execution and search for all instances of `RayView` again.

You'll get a new address this time. Hopefully with the stack logging enabled, you'll get a backtrace for this.

Copy this new address and apply the `msl` command to it.

```
(lldb) msl 0x00007f8250f0a170
```

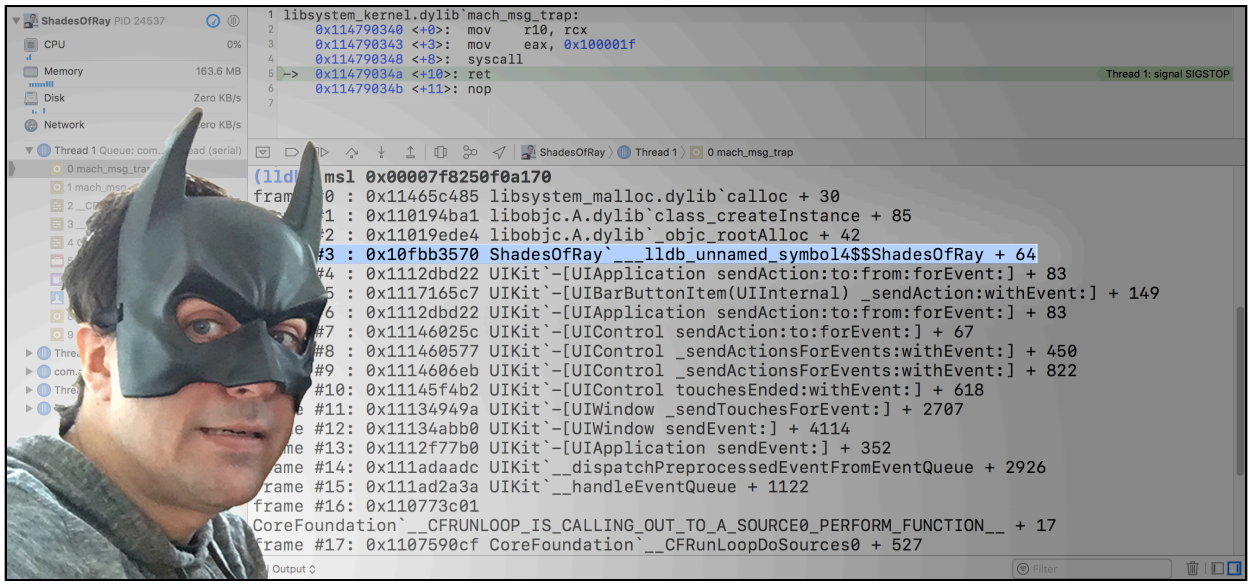
This will give you the stack trace!

```
(lldb) po turn_on_stack_logging(1)
ShadesOfRay(24537,0x1147d43c0) malloc: stack logs being written into /tmp/stack-logs.
24537.11ea53000.ShadesOfRay.g0Y5EW.index
ShadesOfRay(24537,0x1147d43c0) malloc: recording malloc and VM allocation stacks to disk us
standard recorder
0x0000000000000001

(lldb) continue
Process 24537 resuming
(lldb) search RayView -b
RayView * [0x00007f8250e030c0]
RayView * [0x00007f8250f0a170]
(lldb) msl 0x00007f8250f0a170
frame #0 : 0x11465c485 libsystem_malloc.dylib`calloc + 30
frame #1 : 0x110194ba1 libobjc.A.dylib`class_createInstance + 85
frame #2 : 0x11019ede4 libobjc.A.dylib`_objc_rootAlloc + 42
frame #3 : 0x10fbb3570 ShadesOfRay`___lldb_unnamed_symbol4$$ShadesOfRay + 64
frame #4 : 0x1112dbd22 UIKit`-[UIApplication sendAction:to:from:forEvent:] + 83
frame #5 : 0x1117165c7 UIKit`-[UIBarButtonItem(UIInternal) _sendAction:withEvent:] + 149
frame #6 : 0x1112dbd22 UIKit`-[UIApplication sendAction:to:from:forEvent:] + 83
```

This is awesome! You can enable malloc logging at will to monitor any allocation or deallocation events without having to restart your process.

Wait wait wait. Hold on a second... there's a symbol that's stripped.



Ray don't like no stripped functions.

If you recall in the previous chapter, you created the `sbt` command which symbolicated a stack trace. In the `sbt.py` script, you created the `processStackTraceStringFromAddresses` function which took a list of numbers (representing memory addresses for code) and the `SBTarget`. This function then returned a potentially symbolicated string for the stack trace.

You've already done the hard work to write this function, so why not include this work in the `msl.py` script to optionally execute it?

Jump to the very top of the `msl.py` function and add the following import statement:

```
import sbt
```

In the `handle_command` function in **msl.py**, hunt for the following code:

```
retString = sbt.processStackTraceStringFromAddresses(
                                                addresses,
                                                   target)
```

Replace that code with the following:

```
if options.resymbolicate:
    retString = sbt.processStackTraceStringFromAddresses(
                                                    addresses,
                                                       target)
else:
    retString = processStackTraceStringFromAddresses(
                                            addresses,
                                               target)
```

You're conditionally checking for the `options.resymbolicate` option (which I've already set up for you). If `True`, then call the logic in the `sbt` module to see if it can generate a string of resymbolicated functions.

Since you wrote that function to be generic and handle a list of Python numbers, you can easily pass this information from your `msl` script.

Before you test this out, there's one final component to implement. You need to make a convenience command to enable the `turn_on_stack_logging`.

Jump up to the **__lldb_init_module** function (still in `msl.py`) and add the following line of code:

```
debugger.HandleCommand('command alias enable_logging expression -lobjc -O
-- extern void turn_on_stack_logging(int); turn_on_stack_logging(1);')
```

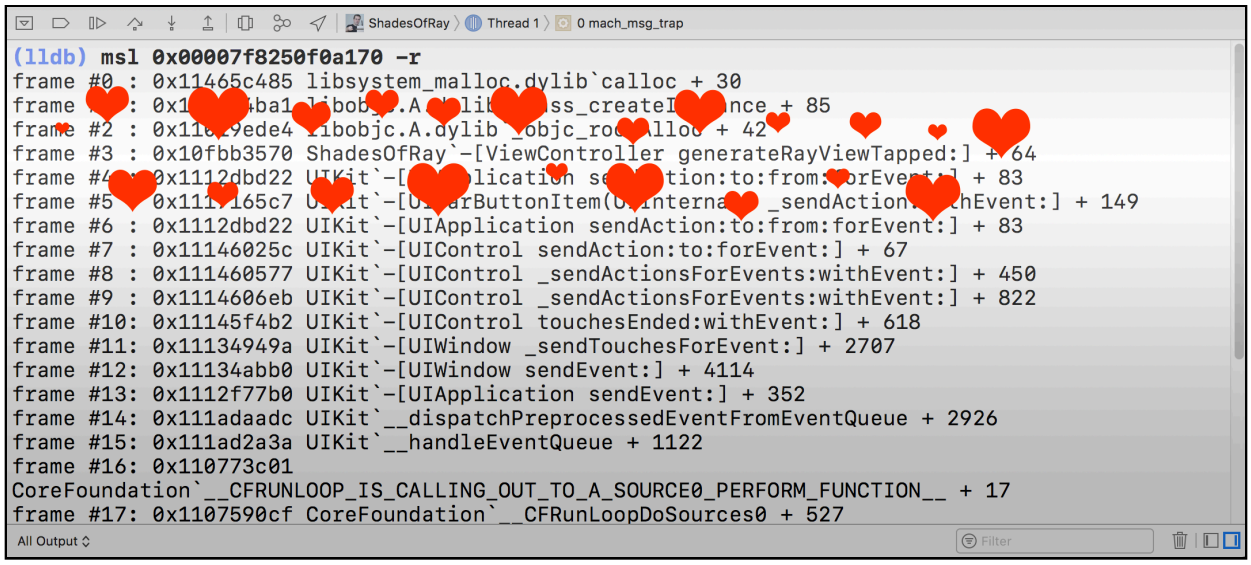
This declares a convenience command to turn on malloc stack logging.

Woot! Done! Jump back to Xcode and reload your script:

```
(lldb) reload_script
```

Use the `--resymbolicate` option on the previous `RayView` to see the stack in its fully symbolicated form.

```
(lldb) msl 0x00007f8250f0a170 -r
```



I am literally crying with happiness in the face of this wholly beautiful stack trace. Snif.

Where to go from here?

Hopefully, this full circle of idea, research & implementation has proven useful and even inspired you to create your own scripts. There's a lot of power hidden quietly away in the many frameworks that already exist on your `[i|mac|tv|watch]OS` device.

All you need to do is find these hidden gems and exploit them for some crazy commercial debugging tools, or even to use in reverse engineering to better understand what's happening.

Here's a list of directories you should explore on your actual iOS device:

- `/Developer/`
- `/usr/lib/`
- `/System/Library/PrivateFrameworks/`

Go forth my little debuggers, and build something that completely blows my mind!

Section V: DTrace

What? Youve never heard of DTrace?! It is AWESOME! DTrace is a tool that lets you explore code in dynamic & static ways.

http://dtrace.org/guide/preface.html

You can create DTrace probes to be compiled into your code (static), or you can inspect any code that is already compiled and running (dynamic). DTrace is a versatile tool: it can be a profiler, an analyzer, a debugger or anything you want.

I often will use DTrace to cast a wide-reaching net over code I want to explore, when I have no clue where I should start.

Chapter 29: Hello, DTrace

Chapter 30: Hello Script Bridging

Chapter 31: DTrace vs. objc_msgSend

Chapter 29: Hello, DTrace

Omagerd! It's **DTrace** time! DTrace is one of the coolest tools you've (likely?) never heard about. With DTrace, you can hook into a function or a group of functions using what's called a **probe**. From there, you can perform custom actions to query information out of a specific process, or even system wide on your computer (and monitor multiple users)!

If you've ever used the Instruments application it might surprise you that a lot of the power underneath it is powered by DTrace.

In this chapter, you'll explore a very small section of what DTrace is capable of doing by tracing Objective-C code in already compiled applications. Using DTrace to observe iOS frameworks (like `UIKit`) can give you an incredible insight into how the authors designed their code.

The bad news

Let's get the bad news out of the way first, because after that it's all exciting and cool things from there. There are several things you need to know about DTrace:

- **You need to disable Rootless for DTrace to work**. Do you remember decades ago in Chapter 1 where I mentioned you need to disable Rootless for certain functionality to work? In addition to letting LLDB attach to any process on your macOS, DTrace will not correctly function if **System Integrity Protection** is enabled. If you skipped Chapter 1, go back and disable Rootless now. Otherwise, you'll need to sit on the sidelines for the remainder of this section.

- **DTrace is not implemented for iOS devices**. Although the Instruments application uses DTrace under the hood for a fair amount of things, it can not run custom DTrace scripts on your iOS device. This means you can only run a limited set of predefined functionality on your iOS device. However, you can still run whatever DTrace scripts you want on the Simulator (or any other application on your macOS) irregardless if you're the owner of the code or not.
- **DTrace has a steep learning curve**. DTrace expects you know what you're doing and what you're querying. The documentation assumes you know the underlying terminology for the DTrace components. You'll learn about the fundamental concepts in this chapter but there is quite literally a whole book on this topic which explores the many aspects of DTrace that are out of the scope of what I'll teach you.

In fact, it's worth noting right up front, if DTrace interests you, get this book http://www.brendangregg.com/dtracebook/index.html. It focuses on a wider range of topics that might not pertain to your Apple debugging/reverse engineering strategies, but it does teach you how to use DTrace.

Now that I've got that off my chest with the bad stuff, it's time to have some fun.

Jumping right in

I am not going to start you off with boring terminology. Ain't nobody got time for that. Instead, you'll first get your hands dirty, then figure out what you're doing later.

Launch the **iPhone X Simulator**. Once alive, create a new **Terminal** window. Type the following into Terminal:

```
sudo dtrace -n 'objc$target:*ViewController::entry' -p `pgrep
SpringBoard`
```

No, this will not secretly destroy your computer, you need that `sudo` in there because DTrace is incredibly powerful and can query information about other users on your computer. This means you need to be `root` to use it.

This DTrace command takes in two options, the **name** option (**-n**) and the PID (**-p**), both of which will be discussed later. Make sure to surround your query in single quotes or else it will not work. Take note of the backtics instead of single quotes that surround `pgrep SpringBoard`.

If you typed out everything correctly, you'll get output in the Terminal window similar to the following:

```
dtrace: description 'objc$target:*ViewController::entry' matched 35794
probes
```

Navigate around the simulator while keeping an eye on the Terminal window.

This will dump out every hit (aka **probe**) that contains the Objective-C class name that ends with "ViewController". Since you left the **function** field blank (don't worry - terminology descriptions are coming in the next section), it matches every single Objective-C method so long as the class name ends with `ViewController`.

Once you get bored of looking at what pops up, kill the Terminal DTrace script with the **Ctrl** + **C** combination.

Back in your Terminal, enter the following:

```
sudo dtrace -n 'objc$target:UIViewController:-viewWillAppear?:entry
{ ustack(); }' -p `pgrep SpringBoard`
```

There's a couple of subtle changes this time:

- The `*ViewController` query has been changed to **UIViewController**.
- The query `-viewWillAppear?` has been added to the **function** location. Again, you'll cover terminology later. For now, all you need to know is instead of matching every function for any class that contains the string "ViewController", this new DTrace script will only match **-[UIViewController viewWillAppear:]**. The question mark stands for a wildcard character in DTrace, which will resolve to the ':' in the `viewWillAppear:` method.
- Finally, you are adding brackets with a function called **ustack()**. This logic will be called every time `-[UIViewController viewWillAppear:]` gets hit. The `ustack()` is one of DTrace's built-in functions which dumps the userland stack trace (aka SpringBoard for this case) when this method gets hit.
- Keep an eye on that single quote which moved from the end of `entry` part to the end of the squiggly bracket.

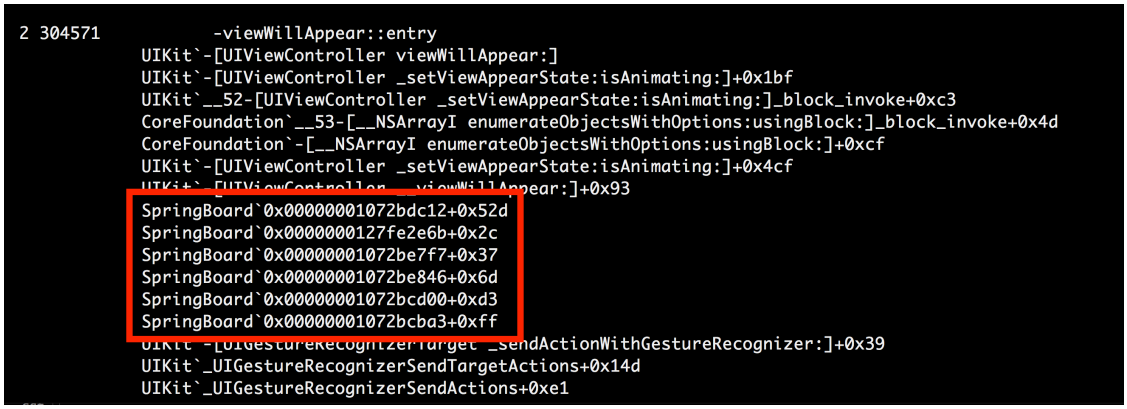
If you typed in everything correctly, you'll get:

```
dtrace: description 'objc$target:UIViewController:-viewWillAppear?:entry
' matched 1 probe
```

Navigate around SpringBoard. Swipe up, swipe down, tap on the **Edit** button by scrolling to the far left, whatever you need to do to trigger a `UIViewController`'s `viewWillAppear:`.

When `UIViewController`'s `viewWillAppear:` gets hit, the stack trace will be printed out in the Terminal.

Take note of some stack traces that don't have the actual function name, but just a module and address.



This is telling us we don't have debugging information or an indirect symbol table to reference the name of this function.

Once you get bored of exploring the stack trace of all the `viewWillAppear:`'s in the SpringBoard process, kill the DTrace script again.

Now... Do you remember the whole spiel about **objc_msgSend** with registers and how the first parameter will be the instance (or class) of an Objective-C class?

For example, when `objc_msgSend` executes, the function signature will look like:

```
objc_msgSend(self_or_class, SEL, ...);
```

You can grab that first parameter (aka the instance of the `UIViewController`) in DTrace with the **arg0** parameter. Unfortunately, you can only get the reference to the pointer - you can't run any Objective-C code, like `[arg0 title]`.

Add the following line of code right before the `ustack()` function in your DTrace command:

```
printf("\nUIViewcontroller is: 0x%p\n", arg0);
```

Your DTrace one-liner will now look like the following:

```
sudo dtrace -n 'objc$target:UIViewController:-viewWillAppear?:entry
{ printf("\nUIViewcontroller is: 0x%p\n", arg0); ustack(); }' -p `pgrep
SpringBoard`
```

Right before printing out the stack trace, you're printing the reference to the `UIViewController` that is calling **viewWillAppear:**.

If you were to copy the address of this pointer spat out by DTrace and attached LLDB to SpringBoard, you will find that it points to a valid `UIViewController` (provided it hasn't been `dealloc`'d yet).

> **Note**: It's easy to get the pointer from `arg0`, but getting any other information (i.e. the class name) is a tricky process.
>
> You can't execute any Objective-C/Swift code in the DTrace script that belongs to the userland process (e.g. SpringBoard). All you can do is traverse memory with the references you have.
>
> In the final chapter, you'll actually get the class name of `arg0` in an Objective-C call by traversing memory in a stripped binary, devoid of debugging information!

Let's do one more DTrace example.

Kill any DTrace scripts and create a script which aggregates all the unique classes that are being executed as you explore SpringBoard:

```
sudo dtrace -n 'objc$target:::entry { @[probemod] = count() }' -p `pgrep
SpringBoard`
```

Navigate around SpringBoard again. You're not going to get any output yet, but as soon as you terminate this script with **Ctrl + C**, you'll get an aggregated list of all the times a method for a particular class was executed. This is called **Aggregations** and you'll learn about this later.

As you can see from my output, SpringBoard had **187075** method calls implemented by `NSObject` that were hit during my run of the above DTrace one-liner.

```
__NSArrayI                                                  11927
NSISVariable                                                12058
__NSDictionaryM                                             13630
NSArray                                                     14663
UIView(ViewGeneration)                                      15034
__NSCFConstantString                                        15619
UIApplication                                               15685
NSThread                                                    16955
UIView                                                      18982
UIGestureEnvironment                                        19813
NSTaggedPointerString                                       23279
UIGestureRecognizer                                         23404
UIKBTree                                                    30476
__NSSetM                                                    47874
OS_object                                                   49878
CALayer                                                     66907
__NSCFString                                               104001
UIView(UIKitManual)                                        114812
__NSArrayM                                                 179334
NSObject                                                   187075
```

It's important to differentiate the fact that these were *very* likely instances of classes which were subclasses of `NSObject` calling methods implemented by `NSObject` (i.e. the subclass of the `NSObject` didn't override any of these methods).

For example, calling `-[UIViewController class]` would count as a hit towards the total methods executed by `NSObject` because `UIViewController` doesn't override the Objective-C method, `class`, nor does `UIViewController`'s parent class, `UIResponder`.

DTrace Terminology

Now that you've gotten your hands dirty on some quick DTrace one-liners, it's time to learn about the terminology so you actually know what's going on in these scripts.

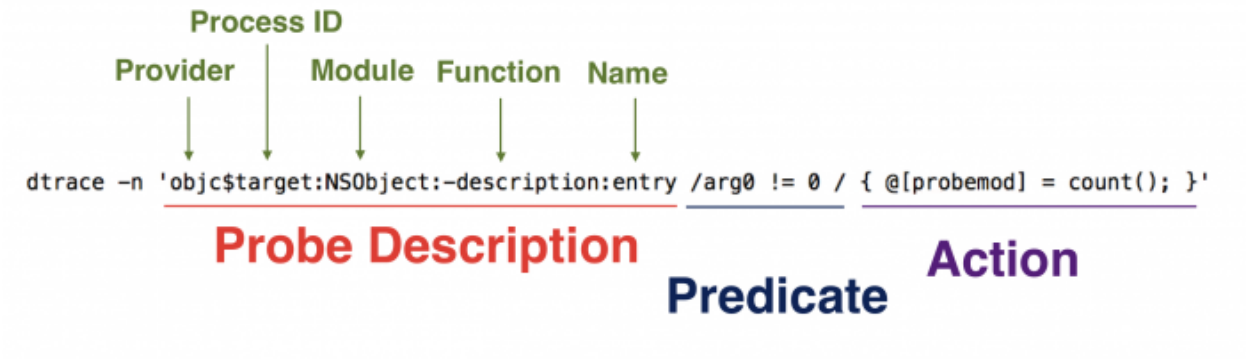
Let's revisit a DTrace **probe**. You can think of a probe as a query. These probes are events that DTrace can monitor either in a specific process or globally across across your computer.

Consider the following DTrace one-liner:

```
dtrace -n 'objc$target:NSObject:-description:entry / arg0 = 0 /
{ @[probemod] = count(); }' -p `pgrep SpringBoard`
```

This example will monitor `NSObject`'s implementation of the `description` method in the process named `SpringBoard`. In addition, this says as soon as the `description` method begins, execute logic to aggregate the amount of times this method is called.

This DTrace one-liner can be further broken down into the following terminology:



- **Probe Description**: Encapsulates a group of items that specify 0 or more probes. This consists of a **provider**, **module**, **function**, and **name**, each separated by colons. Omitting any of these items between the colons will cause the probe description to include all matches. You can use the * or ? operators for pattern matching. The **?** operator will act as a wildcard for a single character, while the * will match anything.
- **Provider**: Think of the provider as a grouping of code or common functionality. For this particular chapter, you'll primarily use the **objc** provider to trace into Objective-C method calls. The `objc` provider groups all of the Objective-C code. You'll explore other providers later.

> **Note**: The **$target** keyword is a special keyword which will match whatever PID you supply DTrace. Certain `providers` (like `objc`) expect you to supply this.
>
> Think of $target as a placeholder for the actual PID, which monitors Objective-C in a specific process. If you do reference the `$target` placeholder, you must specify the target PID through the `-p` or `-c` option flags in your DTrace command.
>
> Typically this is done either by `-p PID` if you knew the exact PID, or more likely `-p` `` `pgrep NameOFProcess` ``. The `pgrep` Terminal command will look for the PID whose process name is `NameOFProcess` then return the PID, which then gets applied to the `$target` variable.

- **Module**: In the `objc` provider, the **module** section is where you specify the class name you wish to observe. Using the `objc` provider is a little unique in this sense, because typically the module is used to reference a library in which the code is coming from. In fact, in some providers, there's no module at all! However, the

authors of the `objc` provider chose to use the module to reference the Objective-C classname. For this particular example, the module is `NSObject`.

- **Function**: The part of the probe description that can specify the function name that you wish to observe. For this particular example, the function is **-description**. The authors of the `objc` provider used the + or - to determine if the Objective-C function is a class or instance method (as you'd expect!). If you changed the function to `+description`, it would query for any probes with `+[NSObject description]` instead.
- **Name**: This typically specifies the location of the probe within a function. Typically, there's the **entry** and **return** names which correspond to a function's entry and exit. In addition, within the `objc` provider, you can also specify any assembly instruction offset to create a probe at! For this particular example, the name is `entry`, or the start of the function.
- **Predicate**: An optional expression to evaluate if the `action` is a candidate for execution. Think of the predicate as the condition in a if-statement. The action section will only execute if the predicate evaluates to true. If you omit the predicate section, then the action block will execute every time for a given probe. For this particular example, the predicate is the `/ arg0 != 0 /`, meaning the content following the predicate will only get evaluated if `arg0` is not `nil`.
- **Action**: The action to perform if the probe matches the probe description and the predicate evaluates to true. This could be as simple as printing something to the console, or performing more advanced functions. For this example, the action is the `@[probemod] = count();` code.

When all of these components are combined, this will form a DTrace **clause**. This consists of the probe description, the optional predicate and optional action.

Put simply, a DTrace clause is made up as follows:

```
provider:module:function:name / predicate / { action }
```

DTrace "one-liners" can comprise multiple clauses which can monitor different items with the probe description, check for different conditions in the predicate and execute different logic with different actions.

So, with the example:

```
dtrace -n 'objc$target:NSView:-init*:entry' -p `pgrep -x Xcode`
```

You have a probe description of `objc$target:NSView:-init*:entry`, which includes `NSView` as the module, `-init*` as the function, and `entry` as the name with no predicate and no action. DTrace produces a default output for tracing (which you can silence with the **-q** option). This default output only displays the `function` and `name`. For example, if you were tracing `-[NSObject init]` without silencing the default DTrace action, your DTrace output would look like the following:

```
dtrace: description 'objc$target:NSObject:-init:entry' matched 1 probe
CPU     ID                    FUNCTION:NAME
  2 512130                      -init:entry
  2 512130                      -init:entry
  2 512130                      -init:entry
  2 512130                      -init:entry
```

From the output, the `-[NSObject init]` got hit 4 times while the process was being traced. You can tell DTrace to use a different formatted output by combining the `-q` option with one of the `print` functions to display alernative formatting for output.

What does that **-n** argument mean again? The `-n` argument specifies the DTrace **name** which can come in the form `provider:module:function:name`, `module:function:name` or `function:name`. In addition, the name option can take an optional probe `clause`, which is why you surround all your one-liner script content in single quotes to pass to the `-n` argument.

Got it? No? You'll repeat the above terminology steps with a useful DTrace option to emphasize what you've learned.

Learning while listing probes

Included in the DTrace command options is a nice little option, **-l**, which will list all the probes you've matched against in your probe description. When you have the **-l** option, DTrace will only list the probes and not execute any actions, regardless of whether you supply them or not.

This makes the `-l` option a nice tool to learn what will and *will not* work.

You will look at a probe description one more time while building up a DTrace script and systematically limiting its scope. Consider the following, Do NOT execute this:

```
sudo dtrace -ln 'objc$target:::' -p `pgrep -x Finder`
```

This will create a probe description on every Objective-C every class, method, and assembly instruction within the **Finder** application. This is a *very bad* idea for a DTrace script and will likely not run on your computer because of the hit count you'll get.

> **Note**: I've supplied the `-x` option to `pgrep` because I could get multiple PIDs for a `pgrep` query, which will screw up the placeholder, `$target`. The `-x` option says only give me the PID(s) that match **exactly** for the name, Finder. If there are multiple instances of a process. You can get the oldest one or newest one in `pgrep` with the `-o` or `-n` option. If this sounds confusing, play around with the `pgrep` command in Terminal without DTrace to understand how it works.

Don't execute the above script because it will take too long. However, execute the rest of these scripts so you understand what's happening.

Let's filter this down a bit. In Terminal, type the following:

```
sudo dtrace -ln 'objc$target:NSView::' -p `pgrep -x Finder`
```

Press enter, then enter your password.

This will list a probe on every single method implemented by `NSView` for all of its methods and every assembly instruction within each of those methods. Still a horrible idea, but at least this one will actually print out after a second.

How many probes is this? You can get that answer by piping your output to the **wc** command:

```
sudo dtrace -ln 'objc$target:NSView::' -p `pgrep -x Finder` | wc -l
```

On my macOS machine in 10.14 (at the time of writing), I get ~42k Objective-C DTrace probes for any code pertaining to `NSView` within the Finder process. Wow!

Filter the probe description down some more:

```
sudo dtrace -ln 'objc$target:NSView:-initWithFrame?:' -p `pgrep -x
Finder`
```

This will filter the probe description down to every assembly instruction that's executed within `-[NSView initWithFrame:]` in addition to the **entry** and **return** probes. Notice the use of a `?` instead of a colon to specify the Objective-C selector (which takes a parameter). If a colon was used, then DTrace will incorrectly parse the input thinking the `function` part was complete and have moved onto specifying the `name` within the DTrace probe. There's also the `-` at the begining of the `function` description to indicate this is an instance Objective-C method.

This is still too much output, you only want to execute a probe that will monitor the beginning of the `-[NSView initWithFrame:]` method and no other parts.

```
sudo dtrace -ln 'objc$target:NSView:-initWithFrame?:entry' -p `pgrep -x Finder`
```

This will say to only set a probe to the beginning of `-[NSView initWithFrame:]` and no other parts in this Objective-C method.

Using the `-l` option is a nice way to learn the scope of your probes before you shoot off making your DTrace actions. I would recommend you make heavy use of the `-l` option when you're starting to learn DTrace.

A script that makes DTrace scripts

When working with DTrace, not only do you get to deal with an exceptionally steep learning curve, you also get to deal with some cryptic errors if you get a build time or runtime DTrace error (yeah, it's on the same level of cryptic as some of those Swift compiler errors).

To help mitigate these build issues as you learn DTrace, I've created a lovely little script called **tobjectivec.py** (trace Objective-C), which is an LLDB Python script that will generate a custom DTrace script for you so long as you ask it real nice like.

> **Note**: Oh yeah, now is a good time to mention you can create DTrace scripts as well as DTrace one-liners. As the complexity in your DTrace logic rises, it becomes a better idea to use a script. For simple DTrace queries, stick with the one-liners.

You'll find the **tobjectivec.py** script located within the starter directory for this chapter. I am assuming you went through Chapter 26, "SB Examples, Improved Lookup" and have installed the **lldbinit.py** script and have stuck it in your `~/lldb` folder. Provided you did this, all you have to do is copy/paste the `tobjectivec.py` script into your `~/lldb` directory and it will be launched next time LLDB starts up.

If you haven't done this yet, go back to Chapter 26 and follow the instructions for installing the `lldbinit.py` file. Alternatively, if you're extremely stubborn, I suppose you can install this `tobjectivec.py` manually by augmenting your `~/.lldbinit` file.

Exploring DTrace through tobjectivec.py

Time to take a whirlwind tour of this script while exploring DTrace on Objective-C code.

Included in the starter folder is the recycled project **Allocator**. Open that project up, build, run, then pause in the debugger.

Once you've got the Allocator project paused, bring up the LLDB console and type the following:

```
(lldb) tobjectivec -g
```

Typically, the `tobjectivec` script will generate a script in the **/tmp/** directory of your computer. However, this **-g** option says that you're debugging your script and displays the output to LLDB instead of creating a file in `/tmp/`. With the `-g` (or `--debug`) option, your current script will be displayed to the console.

This dry run of the `tobjectivec.py` with no extra parameters will produce the following output:

```
#!/usr/sbin/dtrace -s  /* 1 */

#pragma D option quiet  /* 2 */

dtrace:::BEGIN { printf("Starting... use Ctrl + c to stop\n"); } /* 3 */
dtrace:::END   { printf("Ending...\n"  ); }                      /* 4 */

/* Script content below */

objc$target:::entry /* 5 */
{
    printf("0x%016p %c[%s %s]\n", arg0, probefunc[0], probemod,
(string)&probefunc[1]); /* 6 */
}
```

Let's break this down:

1. When executing a DTrace script, the first line needs to be **#!/usr/sbin/dtrace -s** or else the script might not run properly.
2. This line says to not list the probe count nor perform the default DTrace action when a probe fires. Instead, you'll give DTrace your own custom action.
3. This is one third of the DTrace clauses within this script. There are probes for DTrace that monitor for certain DTrace events... like when a DTrace script is about to start. This says, as soon as DTrace starts, print out the `"Starting... use Ctrl + c to stop"` string.

4. Here's another DTrace clause that prints out `"Ending..."` as soon as the DTrace script finishes.
5. This is the DTrace probe description of interest. This says to trace *all* the Objective-C code found in whatever process ID you supply to this script.
6. The action part of this clause prints out the instance of the Objective-C probe that was triggered, followed by Objective-C styled output. In here, you can see **probefunc** and **probemod** being utilized which will be a `char*` representation of the function and module. DTrace has several builtin variables that you can use, `probefunc` & `probemod` being two of them. You also have **probeprov** and **probename** at your disposal. Remember the module will represent the class name while the function will represent the Objective-C method. This takes a combination of the `probemod` & `probefunc` and displays it in the pretty Objective-C syntax you're accustomed to.

Now you've got an idea of this script, remove the `–g` option so you're no longer using the debug option. Type in LLDB:

```
(lldb) tobjectivec
```

You'll get different output this time:

```
Copied script to clipboard... paste in Terminal
```

Your clipboard's contents have been modified. Jump over to your Terminal, then paste in the contents of your clipboard. Here's mine, but yours will of course be different:

```
sudo /tmp/lldb_dtrace_profile_objc.d  –p 95129  2>/dev/null
```

The content you originally saw is now dumped into **/tmp/lldb_dtrace_profile_objc.d**. If you are at all paranoid about what this script does, I recommend you `cat` it first to ensure you know what it's doing.

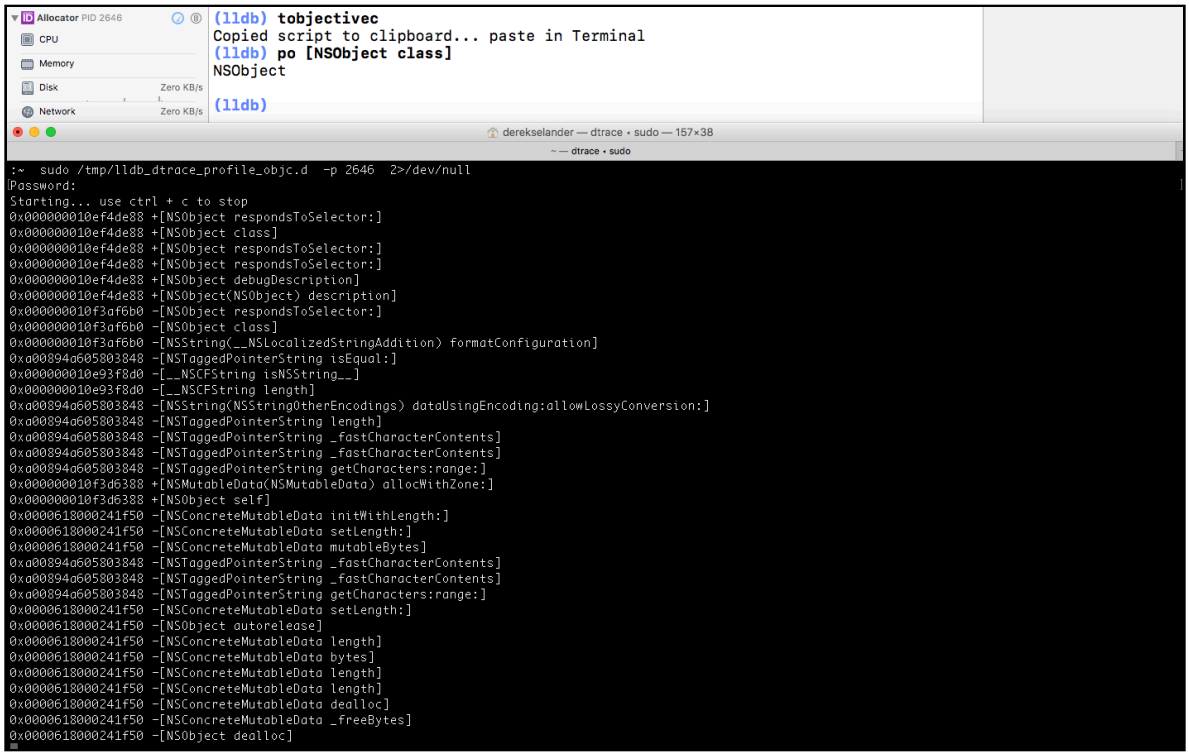
The script provides the process identifier that LLDB is attached to (so you wouldn't have to type `pgrep Allocator`).

Once you get your password prompt, enter in you password to get those root privs:

```
$ sudo /tmp/lldb_dtrace_profile_objc.d  –p 95129  2>/dev/null
Password:
Starting... use Ctrl + c to stop
```

Wait until the DTrace script inidicates to you that it's starting.

With both Xcode and Terminal visible, type a simple `po [NSObject class]` in the console. Check out the slew of Objective-C messages that get spat out for just this method.



This will prepare you for what's about to come. Resume execution using LLDB:

```
(lldb) continue
```

Navigate around the Allocator app (tap on views, bring down the in-call status bar in the Simulator with ⌘ + **Y**) iOS Simulator while keeping an eye on the DTrace Terminal window.

Scary, right?

This is too much stuff. Filter some of the noise by adding content to the module specifier.

Back in Xcode, pause execution of the Allocator process and bring up LLDB.

Generate a new script that only focuses on Objective-C classes that have the phrase `StatusBar` in it's name. Type the following in LLDB:

```
(lldb) tobjectivec -m *StatusBar* -g
```

This will do a dry run and give you the following truncated output:

```
objc$target:*StatusBar*::entry
{
    printf("0x%016p %c[%s %s]\n", arg0, probefunc[0], probemod,
(string)&probefunc[1]);
}
```

Notice how the module portion of the probe has changed. The * can be thought of as `.*` that you know and love in your regular expressions. This means you're querying for probes that contain the case sensitive word `StatusBar` for any Objective-C classes when the probe enters the start of the function.

In LLDB, remove the `-g` option so this script will get copied to your clipboard, then re-execute the command.

```
(lldb) tobjectivec -m *StatusBar*
```

Jump over to your Terminal window. Kill the previous DTrace instance by pressing **Ctrl + C**, then paste in your new script.

```
sudo /tmp/lldb_dtrace_profile_objc.d  -p 2646  2>/dev/null
```

Resume execution back in Xcode.

Jump to the Simulator and toggle the in-call status bar using ⌘ + **Y** or rotate the Simulator by using ⌘ + ← or ⌘ + → while keeping an eye on the DTrace Terminal window.

You'll get a slew of output again.

You can use DTrace to cast a wide net on code with minimal performance hits and quickly drill down when you need to.

Tracing debugging commands

I often find it insightful to know what's happening behind the scenes when I'm executing simple debugging commands and the code that's going on behind them to make it work for me.

Observe how many Objective-C method calls it takes to make a simple Objective-C `NSString`.

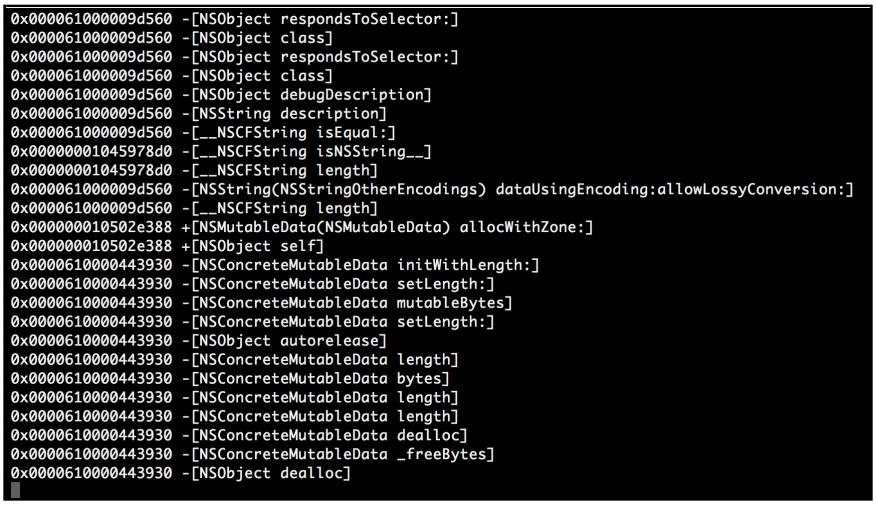
Back in LLDB, type the following:

```
(lldb) tobjectivec
```

Paste the contents in the Terminal window, but do not resume execution in LLDB. Instead, just type the following:

```
(lldb) po @"hi this is a long string to avoid tagged pointers"
```

As soon as you press enter, check out the DTrace Terminal window and see what gets spat out. You'll get something similar to the following:



We just printed out a simple `NSString` and look how many Objective-C calls this took!

Here's one for all you Swift "purists" out there.

Clear the Terminal screen using (⌘ + **K**), make sure the DTrace Terminal script is still running. Head back to LLDB and type the following:

```
(lldb) expression -l swift -O -- class b { }; let a = b()
```

You are using the Swift debugging context to create a pure Swift class then instantiating it. Observe the Objective-C method calls when this class is created.

DTrace will dump out:

```
0x00000001087541b8 +[SwiftObject class]
0x0000000119149778 +[SwiftObject initialize]
0x0000000119149778 +[SwiftObject class]
```

If you were to copy any of the addresses down spat out by DTrace and then `po` that, you'd be greeted with an onslaught of Objective-C method calls for this pure Swift class.

A "pure" Swift class ain't as pure as you thought, right?

Tracing an object

You can use DTrace to easily trace method calls for a particular reference.

Remove the previous DTrace script with **Ctrl** + **C**.

While the application is paused, use LLDB to get the reference to the `UIApplication`. Make sure you are in an Objective-C stack frame.

```
(lldb) po UIApp
```

You'll get something like:

```
<UIApplication: 0x7fa774600f90>
```

Copy the reference and use this to build a predicate which only stops when this reference is arg0 — remember, `objc_msgSend`'s param is a instance of a Class or the Class itself.

```
(lldb) tobjectivec -g -p 'arg0 == 0x7fa774600f90'
```

You'll get the dry run output of your script printed to the console similar to the following:

```
#!/usr/sbin/dtrace -s

#pragma D option quiet
dtrace:::BEGIN { printf("Starting... use Ctrl + c to stop\n"); }
dtrace:::END   { printf("Ending...\n"  ); }

/* Script content below */

objc$target:::entry / arg0 == 0x7fa774600f90 /
{
    printf("0x%016p %c[%s %s]\n", arg0, probefunc[0], probemod,
(string)&probefunc[1]);
}
```

Looks good! Execute the command again without the `-g` option:

```
(lldb) tobjectivec -p 'arg0 == 0x7fa774600f90'
```

Resume execution in LLDB, then paste your script into Terminal.

Trigger the home button, (⌘ + **Shift** + **H**) or the status bar (⌘ + **Y**) in the Simulator.

This is dumping every Objective-C method call on the `[UIApplication sharedApplication]` instance.

Oh, is that too much output to look at? Then aggregate the content!

Back in Xcode, pause execution and in LLDB:

```
(lldb) tobjectivec -g -p 'arg0 == 0x7fa774600f90' -a '@[probefunc] =
count()'
```

This will produce the following script:

```
#!/usr/sbin/dtrace -s

#pragma D option quiet
dtrace:::BEGIN { printf("Starting... use Ctrl + c to stop\n"); }
dtrace:::END   { printf("Ending...\n"  ); }

/* Script content below */

objc$target:::entry / arg0 == 0x7fa774600f90 /
{
    @[probefunc] = count()
}
```

You know the drill. Rerun the above `tobjectivec` command without the `-g` option, then paste your clipboard contents into Terminal and resume execution in LLDB.

No content will be displayed in Terminal yet. But DTrace is quietly aggregating every method that is being sent to the `UIApplication` instance.

Move around in the Simulator to get a healthy count of methods being sent to the `UIApplication`. As soon as you kill this script with the usual **Ctrl + C**, DTrace will dump out the total count of all the Objective-C methods that were applied to the `UIApplication` instance.

Other DTrace ideas

Here's some other ideas for you to try out on your own time:

Trace all the initialization methods for all objects:

```
(lldb) tobjectivec -f ?init*
```

Monitor inter-process communication related logic (i.e. Webviews, keyboards, etc):

```
(lldb) tobjectivec -m NSXPC*
```

Print the `UIControl` subclass which is handling your starting touch event on your iOS device:

```
(lldb) tobjectivec -m UIControl -f -touchesBegan?withEvent?
```

Where to go from here?

This is only the tip of the DTrace iceberg. There's a **lot** more that is possible with DTrace.

I would recommend you check out the following URLs as they are a great resource for learning DTrace.

- https://www.bignerdranch.com/blog/hooked-on-dtrace-part-1/
- https://www.objc.io/issues/19-debugging/dtrace/

In the next chapter, you'll take a deeper dive into whats possible with DTrace and explore profiling Swift code.

Chapter 30: Intermediate DTrace

This chapter will act as a grab-bag of more DTrace fundamentals, destructive actions (yay!), as well as how to use DTrace with Swift. I'll get you excited first before going into theory. I'll start with how to use DTrace with Swift then go into the sleep-inducing concepts that will make your eyes water. Nah, trust me, this will be fun!

In this chapter, you'll learn additional ways DTrace can profile code, as well as how to augment existing code without laying a finger on the actual executable itself. Magic!

Getting started

We're not done picking on Ray Wenderlich. Included in this chapter is yet another movie-title inspired project with Ray's name spliced into it.

Open up the **Finding Ray** application in the **starter** directory for this chapter. No need to do anything special for setup. Build and run the project on the iPhone X simulator.

The majority of this project is written in Swift, though many Swift subclasses inherit from `NSObject` as they need to be visually displayed (if it's an on-screen component, it must inherit from `UIView`, which inherits from `NSObject`, meaning Objective-C)

DTrace is agnostic to whatever Swift code inherits from whatever class as it's all the same to DTrace. You can still profile Objective-C code subclassed by a Swift object so long as it inherits from `NSObject` using the `objc$target` provider. The downside to this approach is if there are any new methods implemented or any overridden methods implemented by the Swift class, you'll not see them in any Objective-C probes.

DTrace & Swift in theory

Let's talk about how one can use DTrace to profile Swift code. There are some pros along with some cons that should be taken into consideration.

First, the happy news: Swift works well with DTrace modules! This means it's very easy to filter out Swift code based on the particular module it's implemented in. The module (aka the `probemod`) will likely be the name of your target in Xcode which contains the Swift code (unless you've changed the target name in Xcode's build settings).

This means you can filter the following Swift code implemented in the `SomeTarget` module like so:

```
pid$target:SomeTarget::entry
```

This will set a probe on the start of every single function implemented inside the `SomeTarget` module. Since the `pid$target` goes after all the non-Objective-C code, this probe will pick C & C++ code as well, but as you'll see in a second, that's easy to filter out with a well-designed query.

Now for the bad news. Since the information about the module is taken up, the Swift classname and function name all go into the DTrace `function` section (aka `probefunc`) for a Swift method. This means you need to be a little more creative with your DTrace querying.

In the previous iteration of Swift (Swift 3), the `probefunc` Swift names returned by DTrace were the mangled Swift names, but that's no longer applicable in Swift 4! DTrace now uses the unmangled Swift names in the output!

So without further ado, let's look at a quick example of a Swift DTrace probe.

Imagine you have a subclass of `UIViewController` named **`ViewController`** which only overrides `viewDidLoad`. Like so:

```
class ViewController: UIViewController {
  override func viewDidLoad() {
    super.viewDidLoad()
  }
}
```

If you want to create a breakpoint on this function, the fullname to this breakpoint would be the following:

```
SomeTarget.ViewController.viewDidLoad() -> ()
```

No surprise there; you've beaten that concept to death in Section 1. If you wanted to search for every `viewDidLoad` implemented by Swift in the `SomeTarget` target (catchy name, right?), you could create a DTrace probe description that looks like the following:

```
pid$target:SomeTarget:*viewDidLoad*:entry
```

This effectively says, "So long as `SomeTarget` and `viewDidLoad` are in the function section, gimme the probe."

Time to try this theory out in the **Finding Ray** application.

DTrace & Swift in Practice

If the **Finding Ray** application is not already running, spark it up. iPhone X Plus Simulator. You know what's up.

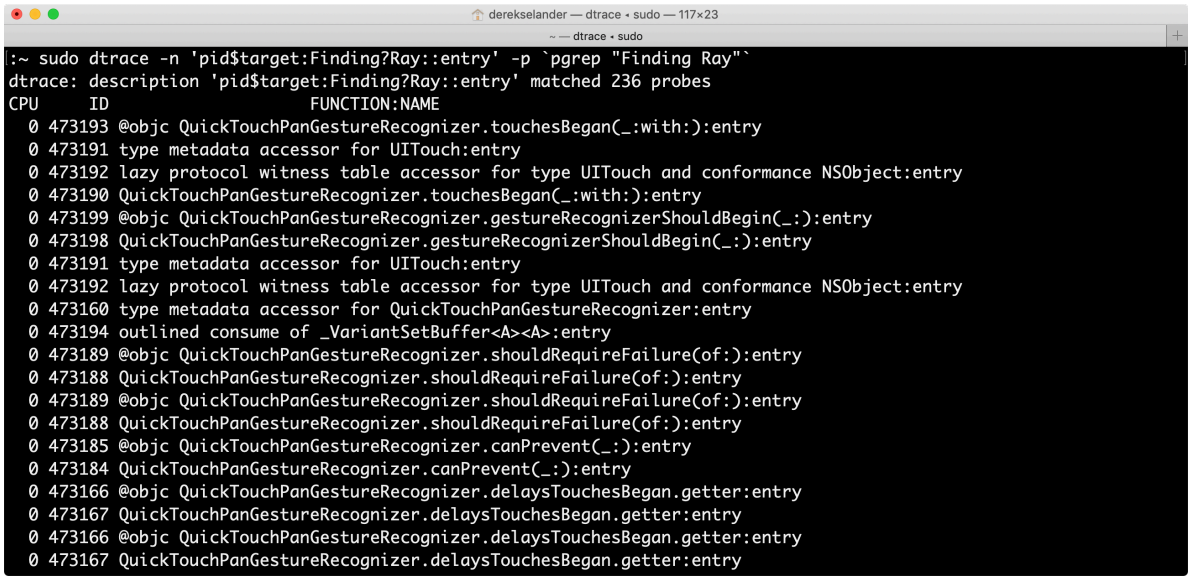
Create a fresh window in `Terminal` and type the following:

```
sudo dtrace -n 'pid$target:Finding?Ray::entry' -p `pgrep "Finding Ray"`
```

I chose an Xcode project name that has a space on purpose. Take note of what you need to do to resolve spaces in an Xcode target when using a DTrace script. The `probemod` section uses a **?** as a placeholder wildcard character for the space. In addition, you need to surround your query when `pgrep`'ing for the process name, otherwise it won't work.

After you've finished typing your password, you'll get ~240 probe `entry` hits for all the non-Objective-C functions inside the **Finding Ray** module.

Click on Ray and drag him around in the Simulator while keeping an eye on all the methods that are getting hit in the `Terminal`.



There's still a bit too much noise. You only want the Swift functions to *only* be displayed. No need to see the probe ID nor the CPU columns.

Kill the DTrace script and replace it with the following:

```
sudo dtrace -qn 'pid$target:Finding?Ray::entry { printf("%s\n",
probefunc); } ' -p `pgrep "Finding Ray"`
```

It's subtle, but you've added the `-q` (or `--quiet`) option. This will tell DTrace to not display the number of probes you've found, nor to display its default output when a probe gets hit. Fortunately, you've also added a `printf` statement to spit out the `probefunc` manually instead.

Wait for DTrace to start up, then drag again.

Much prettier. Unfortunately, you're still getting some methods the Swift compiler generated that I didn't write. You don't want to see any code the Swift compiler has created; you only want to see code I wrote in my Swift classes.

Kill the previous DTrace script and augment this probe description to *only* contain code that you've implemented, and not that of the Swift compiler:

```
sudo dtrace -qn 'pid$target:Finding?Ray::entry { printf("%s\n",
probefunc); } ' -p `pgrep "Finding Ray"` | grep -E "^[^@].*\."
```

Jump over to the Simulator and drag Ray around. Notice the difference?

```
QuickTouchPanGestureRecognizer.delaysTouchesBegan.getter
ViewController.handleGesture(panGesture:)
ViewController.dynamicAnimator.getter
ViewController.snapBehavior.getter
ViewController.containerView.getter
MotionView.animate(isSelected:)
```

This is piping the output to `grep` which is using a regular expression query to say return anything that doesn't contain a "@" and contains a period in the output. This essentially is saying dodn't return any `@objc` bridging methods and a period is gauranteed in any Swift code you write thanks to module namespacing.

One final addition. Augment the script to remove the grep filtering, and instead trace all Swift function entries and exits in the "Finding Ray" module, and use DTrace's **`flowindent`** option.

The flowindent option will properly indent function entries and returns.

```
sudo dtrace -qFn 'pid$target:Finding?Ray::*r* { printf("%s\n",
probefunc); } ' -p `pgrep "Finding Ray"`
```

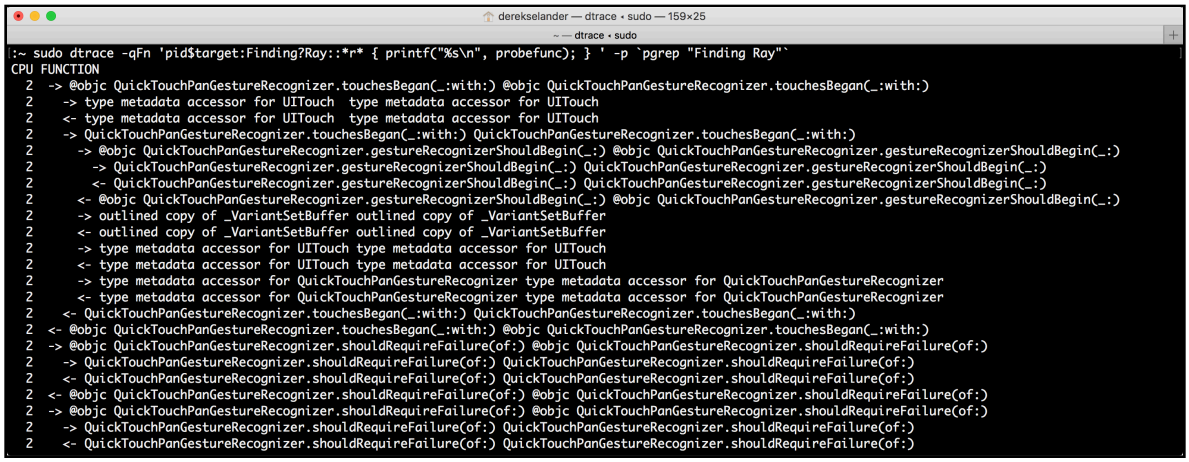
There are a couple of items to note on this one. You've added the `-F` option for `flowindent`. Check out the name section in the probe description, `*r*`. What does this do?

From a DTrace standpoint, most functions in a process have `entry`, `return` and `function` offsets for every assembly instruction. These offsets are given in hexadecimal. This says "give me any name that contains the letter 'r'."

This returns both the `entry` & `return` in the probe description `name`, but omits any function offsets since assembly only goes as high as `f`. Clever, eh?

With both the `enter` & `return` probes of each Swift function enabled, you can clearly see what functions are being executed and where they're being executed from.

Wait for DTrace to start, then drag Ray Wenderlich's face around. You'll get pretty output that looks like this:



Hehehe... thought you would get a kick out of that one!

DTrace variables & control flow

You'll jump into a bit of theory now, which you'll need for the remainder of this section.

DTrace has several ways to create and reference variables in your script. All of them have their own pros and cons as they battle between speed and convenience of use in DTrace.

Scalar variables

The first way to create a variable is to use a **scalar variable**. These are simple variables that can take only take items of fixed size. You don't need to declare the type of scalar variables, or any variables for that matter in your DTrace scripts.

I tend to lean towards using a scalar variable in DTrace scripts to represent a Boolean value, which is due to the limited conditional logic with DTrace — you only have predicates and ternary operators to really branch your logic.

For example, here is a practical case to use a scalar variable:

```
#!/usr/sbin/dtrace -s
#pragma D option quiet

dtrace:::BEGIN
{
    isSet = 0;
    object = 0;
}
objc$target:NSObject:-init:return / isSet == 0 /
{
    object = arg1;
    isSet = 1;
}
objc$target:::entry / isSet && object == arg0 /
{
    printf("0x%p %c[%s %s]\n", arg0, probefunc[0], probemod,
(string)&probefunc[1]);
}
```

This script declares two scalar variables: the **`isSet`** scalar variable will check and see if the **`object`** scalar variable has been set. If not, the script will set the the next object to the `object` variable. This script will trace all Objective-C method calls that are being used on the `object` variable.

Clause-local variables

The next step up are **clause-local** variables. These are denoted by the word **this->** used right before the variable name and can take any type of value, including `char*`'s. Clause-local variables can survive across the *same* probe. If you you try to reference them on a different probe, it won't work. For example, consider the following:

```
pid$target::objc_msgSend:entry
{
  this->object = arg0;
}

pid$target::objc_msgSend:entry / this->object != 0 / {
  /* Do some logic here */
```

```
}

obc$target:::entry {
  this->f = this->object; /* Won't work since different probe */
}
```

I tend to stick with clause-local variables as much as I can since they're quite fast and I don't have to manually free them like I do with the next type of variable…

Thread-local variables

Thread-local variables offer the most flexibility at the price of speed. Additionally, you have to manually release them, otherwise you'll leak memory. Thread-local variables can be used by preceding the variable name with **`self->`**.

The nice thing about thread-local variables is they can be used in different probes, like so:

```
objc$target:NSObject:init:entry {
  self->a = arg0;
}

objc$target::-dealloc:entry / arg0 == self->a / {
  self->a = 0;
}
```

This will assign `self->a` to whatever object is being initialized. When this object is released, you'll need to manually release it as well by setting `a` to 0.

With variables in DTrace out of the way, let's talk about how you can use variables to execute conditional logic.

DTrace conditions

DTrace has extremely limited conditional logic built in. There's no such thing as the if/else-statement in DTrace! This is a conscious decision, because a DTrace script is designed to be fast.

However, it does present a problem for you when you want to conditionally perform logic based upon a particular probe, or information contained within that probe.

To get around this limitation, there are two notable methods you can use to perform conditional logic.

The first workaround is to use a **ternary operator**.

Consider the following contrived Objective-C logic:

```
int b = 10;
int a = 0;

if (b == 10) {
  a = 5;
} else {
  a = 6;
}
```

This can be rewritten in DTrace to use a ternary operator:

```
b = 10;
a = 0;
a = b == 10 ? 5 : 6
```

Here's another example of conditional logic with no else-statement:

```
int b = 10;
int a = 0;
if (b == 10) {
  a++;
}
```

In DTrace form, this would look like:

```
b = 10;
a = 0;
a = b == 10 ? a + 1 : a
```

The other solution to this is to use multiple DTrace clauses along with a predicate. The first DTrace clause will setup the information needed by the second clause to see if it should perform the action in the predicate.

I know you probably forgot all the terminology for these DTrace components so let's also look at an example for this.

For example, let's say you wanted to trace every call in between the start and stop of a function. Typically, I would recommend just setting a DTrace script to catch everything and then use LLDB to execute the command. But what if you wanted to do this solely in DTrace?

For this particular example, you want to trace all Objective-C method calls being executed by `-[UIViewController initWithNibName:bundle:]` with the following DTrace script:

```
#!/usr/sbin/dtrace -s
#pragma D option quiet
```

```
dtrace:::BEGIN
{
  trace = 0;
}

objc$target:target:UIViewController:-initWithNibName?bundle?:entry {
  trace = 1
}

objc$target:target:::entry / trace / {
  printf("%s\n", probefunc);
}

objc$target:target:UIViewController:-initWithNibName?bundle?:return {
  trace = 0
}
```

As soon as the `initWithNibName:bundle:` is entered, the `trace` variable is set. From there on out, every single Objective-C method is displayed until `initWithNibName:bundle:` returns.

Not being able to use loops and conditions can appear annoying at first when writing DTrace scripts, but think of not relying on the common programming idioms you've become accustomed to as a nice brain teaser.

Time for another big discussion: inspecting process memory in your DTrace scripts.

Inspecting process memory

It may come as surprise, but the DTrace scripts you've been writing are actually executed in the kernel itself. This is why they're so fast and also why you don't need to change any code in an already compiled program to perform dynamic tracing. The kernel has direct access!

DTrace has probes all over your computer. There are probes in the kernel, there's probes in userland, there's even probes to describe the crossing between the kernel and userland (and vice versa) using the **`fbt`** provider.

Here's a visualization showing a *very very* small percentage of the DTrace probes on your computer.



Narrow down your focus to just two probes of the thousands by exploring the **`open`** system call and the **`open_nocancel`** system call. Both of these functions are implemented in the kernel and are responsible for any type of file openings for reading, writing, or both.

The system `open` has the following function signature:

```
int open(const char *path, int oflag, ...);
```

Internally, `open` will sometimes call the `open_nocancel`, which has the following function signature:

```
int open_nocancel(const char *path, int flags, mode_t mode);
```

Both of these functions contain a `char*` as the first parameter. You've already grabbed parameters from functions before in DTrace probes using `arg0` and `arg1`.

What you haven't done yet is dereference those pointers to look at their data. Just as in the previous chapters with `SBValue`, you can spelunk in memory with DTrace and even get the string representation of this first parameter in the `open` system calls.

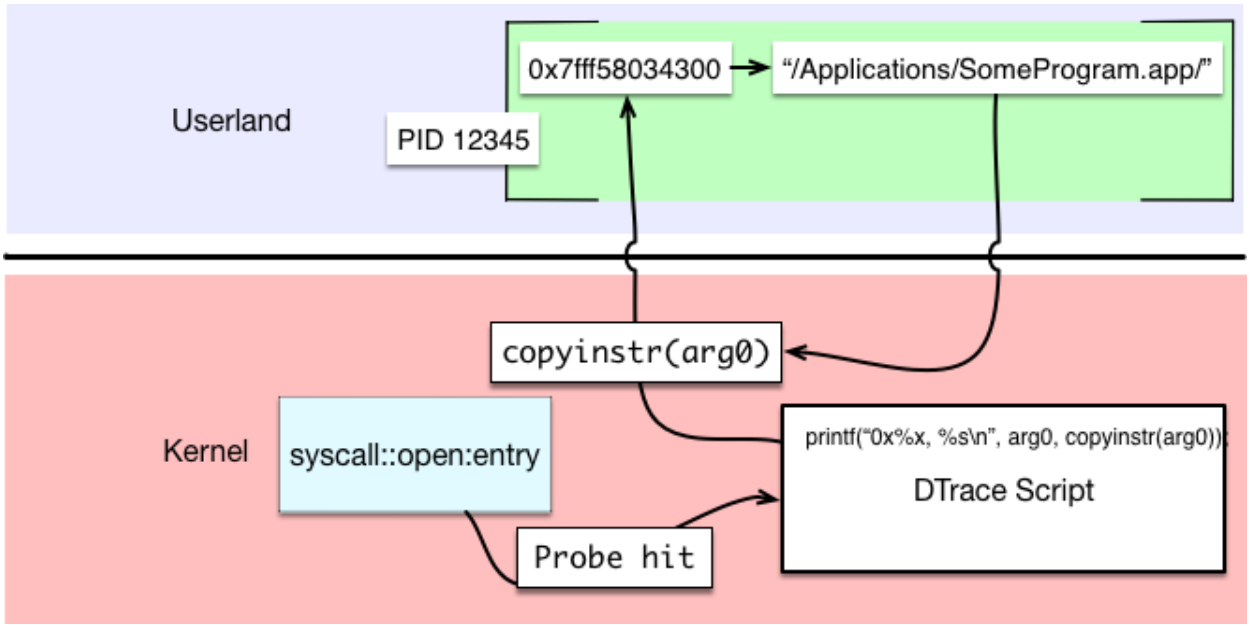
There's one gotcha though. A DTrace script executes in the kernel. The `argX` parameters are given to you, but these are pointers to the value in the address space of the program. However, DTrace runs in the kernel. So you need to manually copy whatever data you're reading into the kernel's memory space.

This is done through the **`copyin`** and **`copyinstr`** functions. `copyin` will take an address with the amount of bytes you want to read, while the `copyinstr` expects to copy a `char*` representation.

In the case of the `open` family of system calls, you could read the first parameter as a string with the following DTrace clause:

```
sudo dtrace -n 'syscall::open:entry { printf("%s", copyinstr(arg0)); }'
```

For example, if a process whose PID was 12345 was attempting to open `"/Applications/SomeApp.app/"`, DTrace could read this first parameter using `copyinstr(arg0)`.



For this particular example, DTrace will read in `arg0`, which for this example equals `0x7fff58034300`. With the `copyinstr` function, the `0x7fff58034300` memory address will be dereferenced to grab the `char*` representation for the pathname, `"/Applications/SomeApp.app/"`.

Playing with open syscalls

With the knowledge you need to inspect process memory, create a DTrace script that monitors the open family of system calls. In `Terminal`, type the following:

```
sudo dtrace -qn 'syscall::open*:entry { printf("%s opened %s\n",
execname, copyinstr(arg0)); ustack(); }'
```

This will print the contents of `open` (or `open_nocancel`) along with the program that called the `open*` system call with the userland stack trace that was responsible for the call.

Isn't DTrace awesome!?

Augment your `open` family of system calls to only focus on the **Finding Ray** process.

```
sudo dtrace -qn 'syscall::open*:entry / execname == "Finding Ray" /
{ printf("%s opened %s\n", execname, copyinstr(arg0)); ustack(); }'
```

> **Note**: The actions you perform with DTrace can sometimes produce errors to `stderr` in Terminal. Depending on the error, you can get around this by creating checks for appropriate input with a DTrace `predicate`, or you can filter your `probe description` query with less probes. An alternative to this is to ignore all errors produced by DTrace by adding `2>/dev/null` in your DTrace one-liner. This effectively tells your DTrace one-liner to pipe any `stderr` content (2 is the standard error file descriptor) to be ignored. I often use this solution to cast a wide net on probes that can be error-prone, but ignore any errors that my tracing produces.

Rebuild an launch the application.

Stack traces will now only be displayed on any `open*` system call being called from the **Finding Ray** application. Play around with the app in the Simulator a bit and see if you can make it output something!

Filtering open syscalls by paths

Inside the **Finding Ray** project, I remember I used the image named `Ray.pdf` for something, but I can't remember where. Good thing I have DTrace along with `grep` to hunt down the location of where `Ray.pdf` is being opened.

Kill your current DTrace script and modify the script so it pipes `stderr` straight to hell. While you're doing that, append a `grep` query to it so it looks like:

```
sudo dtrace -qn 'syscall::open*:entry / execname == "Finding Ray" /
{ printf("%s opened %s\n", execname, copyinstr(arg0)); ustack(); }' 2>/
dev/null | grep Ray.png -A40
```

This pipes all stderr to nowhere, stdout to `grep` and searches for any references to the `Ray.png` image. If there's a hit, print out the next 40 lines.

> **Note**: There's actually a pretty awesome DTrace script called **`opensnoop`** found in `/usr/bin/` on your computer which has many options for monitoring the `open` family of system calls and is wayyyyyyyy easier to use than writing these scripts. But you wouldn't learn anything if I just gave you the easy way out, right? Check out this script on your own time, with a good ol' `man opensnoop`. You won't be disappointed in what it can do.

There's a more elegant way to do this without relying on piping (well, more elegant in *my* opinion). You can use the predicate section of the DTrace clause to search the userland `char*` input for the `Ray.png` string.

You'll use the **`strstr`** DTrace function to do this check. This function takes two strings and returns a pointer to the first occurrence of the second string in the first string. If it can't find an occurrence, it will return `NULL`. This means you can check if this function equals `NULL` in the predicate to search for a path which contains `Ray.png`!

Augment your increasingly ugly — er, *complex* DTrace script to look like the following:

```
sudo dtrace -qn 'syscall::open*:entry / execname == "Finding Ray" &&
strstr(copyinstr(arg0), "Ray.png") != NULL / { printf("%s opened %s\n",
execname, copyinstr(arg0)); ustack(); }' 2>/dev/null
```

Build and rerun the application.

You threw out the `grep` piping and replaced it with a conditional check in the predicate for anything containing the name `Ray.png` that's opened in the **Finding Ray** process.

In addition, you've easily pinpointed the stack trace responsible for opening the Ray.png image.

DTrace & destructive actions

> **Note**: What I am about to show you is very dangerous.
>
> Let me repeat that: This next bit is very dangerous.
>
> If you screw up a command you could lose some of your beloved images. Follow along only at your own risk!
>
> In fact, to be safe, please close any applications that pertain to using photos (i.e. Photos, PhotoShop, etc). Neither I, nor the publisher are legally responsible for anything that could happen on your computer.
>
> You have been warned!

Heh... I bet that above legal section made you nervous.

You'll use DTrace to perform a **destructive action**. That is, normally DTrace will only monitor your computer, but now you'll actually alter logic in your program.

You'll monitor the `open` family of system calls that are executed by the **Finding Ray** app. If one of the `open` system calls contain the phrase `.png` in its first parameter (aka the parameter of type `char*` to the path it's opening), you'll replace that argument with a different PNG image.

This can all be accomplished with the **`copyout`** and **`copyoutstr`** DTrace commands. You'll use the `copyoutstr` explicitly for this example. You'll notice these name are similar to `copyin` and `copyinstr`. The `in` and `out` in this context refer to the direction in which you're copying data, either into where DTrace can read it, or out to where the process can read it.

In the **projects** directory, there's a standalone image named **troll.png**. Create a new window in Finder with ⌘ + **N**, then navigate to your home directory by pressing ⌘ + **Shift** + **H**. Drop **troll.png** into this directory (feel free to remove it when this chapter is done). There's a method to this madness — just bear with me!

Why did you need to do this? You're about to write to memory in an existing program. There's only a finite amount of space that is already allocated for this string in the program's memory. This will likely be some long string because you're in the iPhone Simulator and your process (mostly) reads images found in its own sandbox.

Do you remember searching for **Ray.png**? Here's that full path on my computer. Yours will obviously be different.

```
/Users/derekselander/Library/Developer/CoreSimulator/Devices/
97F8BE2C-4547-470C-955F-3654A8347C41/data/Containers/Bundle/Application/
102BDE66-79CB-453C-BA71-4062B2BC5297/Finding Ray.app/Ray.png
```

The plan of attack is to use DTrace with a shorter path to an image, which will result in something like this in the program's memory:

```
/Users/derekselander/troll.png\0veloper/CoreSimulator/Devices/
97F8BE2C-4547-470C-955F-3654A8347C41/data/Containers/Bundle/Application/
102BDE66-79CB-453C-BA71-4062B2BC5297/Finding Ray.app/Ray.png
```

You see that `\0` in there? That's the `NULL` terminator for `char*`. So essentially this string is really just:

```
/Users/derekselander/troll.png
```

Because that's how `NULL` terminated strings work!

Getting your path length

When writing data out, you'll need to figure out how many `chars` your fullpath is to the `troll.png`. I know the length of mine, but unfortunately, I don't know your name nor the name of your computer's home directory.

Type the following in `Terminal`:

```
echo ~/troll.png
```

This will be dump the fullpath to the troll.png image. Hold onto this for a second as you'll paste this into your script. Also figure out how many characters this is in `Terminal`:

```
echo ~/troll.png | wc -m
```

In my case, `/Users/derekselander/troll.png` is 31 `char`'s. But here's the gotcha: *You need to account for the null terminator*. This means the total length I need to insert my new string needs to be an existing `char*` of length 32 or greater.

The `arg0` in `open*` is pointing to something in memory. If you were to write in this location with something longer than this string, then this could corrupt memory and kill the program. Obviously, you don't want this, so what you'll do is stick **troll.png** in a directory that has a shorter character count.

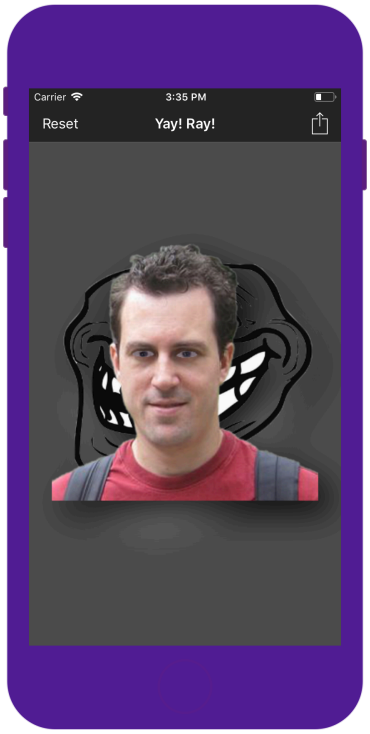
You'll also perform checks via the DTrace predicate to ensure you have enough room as well. C'mon, you're a thorough and diligent programmer, right?

Type the following in `Terminal`, replacing `/Users/derekselander` and `32` with your values:

```
sudo dtrace -wn 'syscall::open*:entry / execname == "Finding Ray" && arg0
> 0xfffffffe && strstr(copyinstr(arg0), ".png") != NULL &&
strlen(copyinstr(arg0)) >= 32 / { this->a = "/Users/derekselander/
troll.png"; copyoutstr(this->a, arg0, 32); }'
```

Rebuild and run **Finding Ray** while this new DTrace script is active.

Provided you've executed everything correctly, each time the **Finding Ray** process tries to open a file that contains the phrase ".png", you'll return troll.png instead.



Other destructive actions

In addition to `copyoutstr` and `copyout`, DTrace has some other destructive actions worth noting:

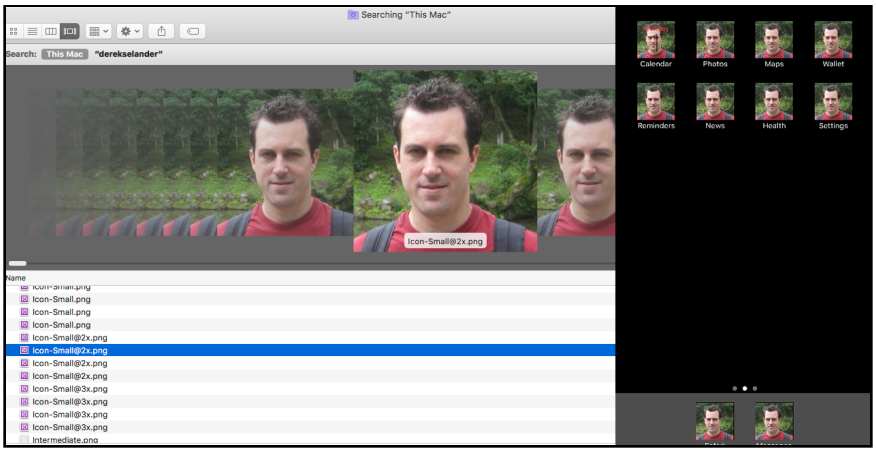
- **stop(void)**: This will freeze the currently running *userland* process (given by the **pid** built-in argument). This is ideal if you want to stop execution of a userland program, attach LLDB to it and explore it further.
- **raise(int signal)**: This will raise a signal to the process responsible for a probe.
- **system(string program, ...)**: This lets you execute a command just as if you were in `Terminal`. This has the added benefit of letting you access all the DTrace built-in variables, such as `execname` and `probemod`, to use in a `printf`-style formatting.

I encourage you to explore these destructive actions (especially the `stop()` action) on your own time. That being said, be careful with that `system` function. You can do a lot of damage really easily if used incorrectly.

Where to go from here?

There are many powerful DTrace scripts on your macOS machine. You can hunt for them using the `man -k dtrace`, then systematically `man`'ing what each script does. In addition, you can learn a lot by studying the code in them. Remember, these are *scripts*, not compiled executables, so source-code is fair game.

Also, be very careful with destructive actions. That being said, you can put Ray Wenderlich *everywhere on your computer*:



Isn't that what you've always wanted?

In all seriousness, you can do some pretty crazy stuff to your computer and gain a lot of insight using DTrace.

Chapter 31: DTrace vs. objc_msgSend

You've seen how powerful DTrace is against Objective-C and Swift code which you have the source for, or code that resides in a Framework like `UIKit`. You've used DTrace to trace this code and make interesting tweaks all while performing zero modifications to already compiled source code.

Unfortunately, when DTrace is put up against a stripped executable, it is unable to create any probes to dynamically inspect those functions.

However, when exploring Apple code, you still have one very powerful ally on your side: **`objc_msgSend`**. In this chapter you'll use DTrace to hook `objc_msgSend`'s **entry** probe and pull out the class name along with the Objective-C selector for that class.

By the end of this chapter, you'll have LLDB generating a DTrace script which only generates tracing info for code implemented within the main executable that calls **`objc_msgSend`**.

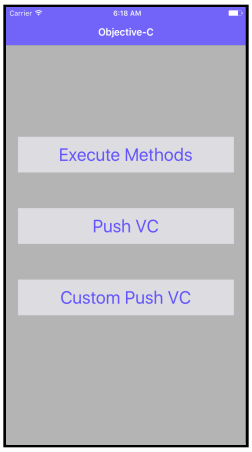
Building your proof-of-concept

Included in the **starter** folder is an app called **VCTransitions**, which is a very basic Objective-C/Swift application that showcases a normal `UINavigationController` push transition, as well as a custom push transition.

Open up this Xcode project, build and run on the iPhone XS Simulator and take a quick look around.

It's important to note, there are two schemes inside this application: **VCTransitions** and **Stripped VCTransitions**. Make sure to select the *VCTransitions* scheme when running. We'll talk more about the Stripped VCTransitions scheme in a second.

> **Note**: Normally I don't care about the exact version of the software you're running, so long as it's iOS 12. This time, however, I insist you run **iOS 12.1.x (or earlier)** since you'll be viewing assembly that could change in a future release. You'll be exploring some assembly in this chapter, and I can't guarantee it's unchanged in a new iOS version that I've not viewed (at the time of writing).



There are buttons to perform the two navigation pushes, and there's also a button named **Execute Methods** that will loop through all known Objective-C methods which are implemented/overriden by a given `Class`. If the method takes no parameters, it executes it.

For example, the first view controller displayed is `ObjCViewController`. If you tap **Execute Methods**, it will call **`anEmptyMethod`** as well as all the getters for the overridden properties, since all of those methods don't require parameters.

Now, onto the fun stuff.

Jump over to **OjbCViewController.m** and take a look at the `IBAction` methods implemented by this class.

Make a DTrace one-liner in Terminal to ensure that you can see these methods getting hit.

Make sure the Simulator is alive and running the **VCTransitions** project.

In Terminal:

```
sudo dtrace -n 'objc$target:ObjCViewController::entry' -p `pgrep
VCTransitions`
```

Press Enter to start this bad boy up. Enter your password when DTrace asks you then head back over to the Simulator and start tapping on buttons. You'll see the `Terminal`

DTrace window fill up with the `IBAction` methods implemented by `ObjCViewController`.



Now, tap one of the **push** buttons so you're on the **SwiftViewController** view controller.

Although this is a subclass of `UIViewController`, tapping on the `IBActions` will not produce any results for the **objcPID** probe. Even though there are dynamic methods implemented or overridden by `SwiftViewController`, *and* being executed through `objc_msgSend`, the actual code is Swift code (even those `@objc` bridging methods).

Pop quiz: If `SwiftViewController` contains the following code:

```
class SwiftViewController: UIViewController,
UIViewControllerTransitioningDelegate {
  @objc var coolViewDTraceTest: UIView? = nil
  @objc var coolBooleanDTraceTest: Bool = false

  // ...
```

Will an Objective-C DTrace probe pick up `coolBooleanDTraceTest` or `coolViewDTraceTest`?

To answer this, first see if these Swift properties are even exposed as Objetive-C probes. They should be, right? They have the `@objc` attributes.

Type the following in Terminal:

```
sudo dtrace -ln 'objc$target::*cool*Test*:entry' -p `pgrep VCTransitions`
```

Dang, only the properties for the Objective-C `ObjCViewController` are displayed and not `SwiftViewControllers`! This is because of Swift proposition 160 https://github.com/apple/swift-evolution/blob/master/proposals/0160-objc-inference.md, which includes a proposition that `NSObject`'s no longer infer `@objc`. In addition, Swift will not create an Objective-C symbol even for dynamic code.

This means you'll have to use the non-Objective-C provider to query Swift DTrace probes.

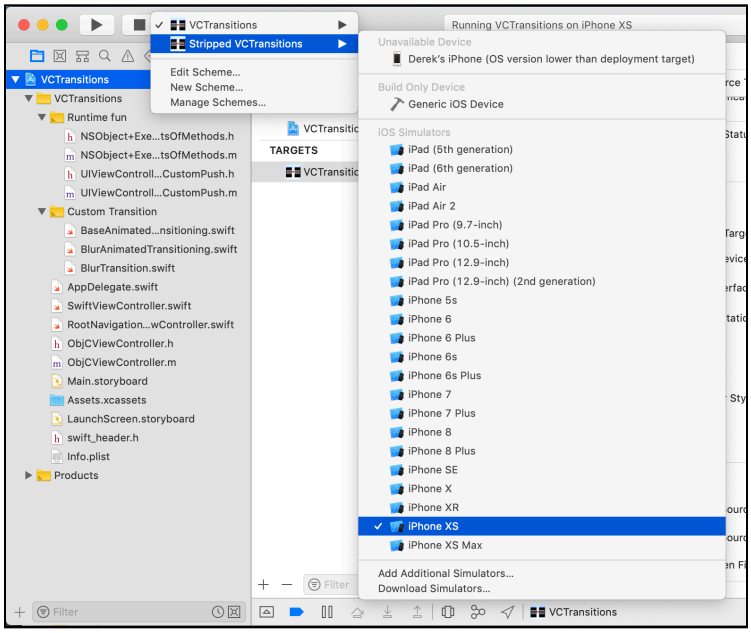
You can confirm this by augmenting your DTrace script to dump *any* methods that include the word *cool* followed sometime later by the word *Test*, like so:

```
sudo dtrace -n 'pid$target::*cool*Test*:entry' -p `pgrep VCTransitions`
```

This is another reason to go after `objc_msgSend` instead of the `objc$target` probe, because calls to `objc_msgSend` will catch dynamically executed Swift code, where objc$target will miss them.

Repeating your steps on a stripped build

Included within the project is a scheme called **Stripped VCTransitions**.



This runs the exact same target (executable) as the **VCTransitions** app, except Xcode will generate a stripped build that doesn't contain any debugging information.

Select the **Stripped VCTransitions** scheme, make sure it's on the iPhone XS Simulator (again on iOS 12 or earlier) and build and run.

Once running, pause the application and bring up LLDB. Search for any code that belongs to `SwiftViewController` using your newly created `image lookup` alternative, `lookup` command, you created in Chapter 26, "SB Examples, Improved Lookup" (if you skipped that chapter, default back to using `image lookup -rn`).

```
(lldb) lookup SwiftViewController
```

Hmm... you won't get any hits. Maybe it's a Swift bug? Try dumping everything pertaining to **ObjCViewController**:

```
(lldb) lookup ObjCViewController
```

Still nothing. What gives?

This executable has been **stripped** of it's information. You can't use the debugging symbols typically available to you to reference an address in memory.

However, LLDB *is* smart enough to realize these locations in memory are, in fact, functions. LLDB will generate a unique function name for the methods it doesn't have information for. The automatically generated function name will take the following form:

```
___lldb_unnamed_symbol[FUNCTION_ID]$$[MODULE_NAME]
```

This means you can list all the functions LLDB has generated inside the **VCTransitions** executable with the following `lookup` command:

```
(lldb) lookup VCTransitions
```

I get 292 hits, with the following truncated output:

```
...
___lldb_unnamed_symbol289$$VCTransitions

___lldb_unnamed_symbol290$$VCTransitions

___lldb_unnamed_symbol291$$VCTransitions
```

Dang, LLDB can't get the names of these functions. Do you think DTrace can read content in a stripped binary?

Type the following in `Terminal`:

```
sudo dtrace -ln 'objc$target:ObjCViewController::' -p `pgrep
VCTransitions`
```

This queries the **VCTransitions** process for the count of probes containing the module `ObjCViewController`, which is DTrace's way of referencing an Objective-C class.

I get the following:

```
   ID   PROVIDER            MODULE                          FUNCTION NAME
dtrace: failed to match objc57009:ObjCViewController:: No probe matches
description
```

I can tell my PID is 57009 and I get 0 hits!

If I wanted to ensure that `ObjCViewController` was producing valid probes (which you saw earlier), simply rebuild this project using the non-stripped Xcode scheme, then run the above Terminal command again. I'll leave that exercise to you if you're interested in proving this works.

How to get around no probes in a stripped binary

So how can you architect a DTrace action and/or probe to get around this hurdle of not being able to inspect a stripped binary?

Since you know Objective-C (and dynamic Swift) methods need to go through `objc_msgSend` (or similar for super calls), you can use the knowledge you've learned about `objc_msgSend` to figure out how to create a nice DTrace action that prints out the name of the class along with the Objective-C selector.

A quick reminder about how `objc_msgSend` works. The function signature looks like this:

```
objc_msgSend(instance_or_class, SEL,  ...);
```

So, `objc_msgSend` takes a class or instance as the first parameter, the Objective-C selector as the second, followed by a variable amount of arguments.

With that in mind, if you had the following code:

```
UIViewController *vc = [UIViewController new];
[vc setTitle:@"yay, DTrace"];
```

The compiler would translate it into the following pseudocode:

```
vc = objc_msgSend(UIViewControllerClassRef, "new");
objc_msgSend(vc, "setTitle:", @"yay, DTrace");
```

From a DTrace standpoint, getting the Objective-C selector is rather easy. Just `copyinstr(arg1)` and you're golden. As you've learned earlier, this will copy the pointer from `arg1`, the Objective-C selector (aka a `char*`), into kernel-land so DTrace can read it.

Now for the hard part: You want the class name of the first parameter passed into `objc_msgSend` as a `char*`.

DTrace won't let you execute arbitrary methods, so you can't rely on the Objective-C runtime, or any of the methods it implements, to dig the information out for you.

Instead, you get to go spelunking through the memory of the `arg0` instance and find the `char*` yourself, which represents the class name, then automate it into a DTrace script.

Hey, this is the culmination of your DTrace skills coming together! You might as well go all out.

Researching method calls using... DTrace!

Let's see if there are any documented ways to go after this thing. In the **objc/runtime.h** header, you have the following declaration:

```
struct objc_class {
    Class isa  OBJC_ISA_AVAILABILITY;

#if !__OBJC2__
    Class super_class
OBJC2_UNAVAILABLE;
    const char *name
OBJC2_UNAVAILABLE;
    long version
OBJC2_UNAVAILABLE;
    long info
OBJC2_UNAVAILABLE;
    long instance_size
OBJC2_UNAVAILABLE;
    struct objc_ivar_list *ivars
OBJC2_UNAVAILABLE;
    struct objc_method_list **methodLists
OBJC2_UNAVAILABLE;
    struct objc_cache *cache
OBJC2_UNAVAILABLE;
    struct objc_protocol_list *protocols
OBJC2_UNAVAILABLE;
#endif

} OBJC2_UNAVAILABLE;
/* Use `Class` instead of `struct objc_class *` */
```

Back in the Objective-C 2.0 days with a 64-bit machine, if you had a pointer at `X` which pointed to a valid class, you could get to that `const char *name` described in the `#if !__OBJC2__`:

```
po *(char *)(X + 0x10)
```

Unfortunately, this is rather dated. This class structure dates back to before Objective-C 2.0. Structs and pointer locations have long since changed. Apple has opted to make the current layout of the `objc_class` structs a little less public for your viewing pleasure.

This means you need to hunt for a function that takes an Objective-C class (or instance of the class) and returns a `char*` for the class so we can figure out what it's doing.

> **Note**: Although hidden in the headers from developers, you *can* obtain the Objective-C class layout by navigating to the latest version here https://opensource.apple.com/source/objc4/ and hunting for the **objc_class** struct. However, for this tutorial, you'll determine what you need to do by looking at the assembly instead of this C struct.

Fortunately, jumping back to the `objc/runtime.h` header file, there's also a function by the name of **`class_getName`**. Don't believe me? Execute `open -h runtime.h` in Terminal.

Looking at the headerfile, `class_getName` has the following signature:

```
/**
 * Returns the name of a class.
 *
 * @param cls A class object.
 *
 * @return The name of the class, or the empty string if \e cls is \c
Nil.
 */
OBJC_EXPORT const char *class_getName(Class cls)
    OBJC_AVAILABLE(10.5, 2.0, 9.0, 1.0);
```

This function takes a `Class` and returns a `char*`. You'll use DTrace to trace this method and see what methods this class is calling underneath the covers.

Hopefully, your **VCTransitions** app is still running. If not, re-run the application. Once active, pause the application in LLDB.

Get the reference to the `Class` representing a `UIView`:

```
(lldb) p/x [UIView class]
```

You'll get something similar:

```
(Class) $0 = 0x0000000109d4ce60 UIView
```

Take this reference to the `UIView` class and apply it to the `class_getName` function:

```
(lldb) po class_getName(0x0000000109d4ce60)
```

You'll get a number? Why is that?

```
0x000000010999319f
```

Oh yeah, duh... this function returns a C `char*`. You have to cast those:

```
(lldb) po (char *)class_getName(0x0000000109d4ce60)
```

You'll now use DTrace to trace all the non Objective-C methods `class_getName` calls behind the scenes.

Jump over to a fresh Terminal session and execute the following DTrace one-liner:

```
sudo dtrace -n 'pid$target:::entry' -p `pgrep VCTransitions`
```

All the while, LLDB should still be suspended when setting up your DTrace script.

Jump on back to LLDB and re-execute that `class_getName` function with the reference to the `UIView` class. Your pointer to the `UIView` class will be different:

```
(lldb) po (char *)class_getName(0x0000000109d4ce60)
```

After you've executed the above command, the DTrace script will spit out the following list of functions that were called for `class_getName`.

```
:~ sudo dtrace -n 'pid$target:::entry' -p `pgrep VCTransitions`
Password:
dtrace: description 'pid$target:::entry' matched 901911 probes
CPU     ID                    FUNCTION:NAME
  6 1405417              class_getName:entry
  6 1405416 objc_class::demangledName(bool):entry
  6 566986        _NSPrintForDebugger:entry
  6 1405847               objc_msgSend:entry
```

It looks like that `objc_class::demangledName(bool):` function is a fun place to explore.

Kill the DTrace script. You don't want it screwing with your LLDB breakpoints, as setting a DTrace probe on a LLDB breakpoint can have unexpected consequences.

Once the DTrace script has terminated, set a breakpoint on `objc_class::demangledName(bool)` with LLDB, like so:

```
(lldb) b objc_class::demangledName(bool)
```

Rerun the expression, but tell LLDB to honor breakpoints:

```
(lldb) exp -i0 -O -- class_getName([UIView class])
```

As soon as you press enter, LLDB will stop on this `objc_class::demangledName(bool)` function.Take a good look at the assembly.

```
libobjc.A.dylib`objc_class::demangledName:
->  0x112cdde4a <+0>:   push   rbp
    0x112cdde4b <+1>:   mov    rbp, rsp
    0x112cdde4e <+4>:   push   r15
    0x112cdde50 <+6>:   push   r14
    0x112cdde52 <+8>:   push   r13
    0x112cdde54 <+10>:  push   r12
    0x112cdde56 <+12>:  push   rbx
    0x112cdde57 <+13>:  sub    rsp, 0x28
    0x112cdde5b <+17>:  mov    r12d, esi
    0x112cdde5e <+20>:  mov    r15, rdi
    0x112cdde61 <+23>:  movabs r13, 0x7ffffffffff8
    0x112cdde6b <+33>:  mov    rax, qword ptr [r15 + 0x20]
    0x112cdde6f <+37>:  and    rax, r13
    0x112cdde72 <+40>:  cmp    dword ptr [rax], 0x40000000
    0x112cdde78 <+46>:  jb     0x112cdde8b               ; <+65>
    0x112cdde7a <+48>:  mov    rbx, qword ptr [rax + 0x38]
    0x112cdde7e <+52>:  test   rbx, rbx
    0x112cdde81 <+55>:  jne    0x112cddf80               ; <+310>
    0x112cdde87 <+61>:  mov    rax, qword ptr [rax + 0x8]
    0x112cdde8b <+65>:  add    rax, 0x18
    0x112cdde8f <+69>:  mov    rbx, qword ptr [rax]
    0x112cdde92 <+72>:  xor    esi, esi
    0x112cdde94 <+74>:  mov    rdi, rbx
    0x112cdde97 <+77>:  call   0x112cdbab7               ; copySwiftV1DemangledName(char const*, bool)
    0x112cdde9c <+82>:  mov    r14, rax
    0x112cdde9f <+85>:  mov    rcx, qword ptr [r15 + 0x20]
    0x112cddea3 <+89>:  and    rcx, r13
    0x112cddea6 <+92>:  cmp    dword ptr [rcx], 0x40000000
    0x112cddeac <+98>:  jb     0x112cddede               ; <+148>
```

Scary assembly, part I

As always, this stuff looks scary at first. But when you systematically go through it, it's not that bad. You'll actually break the assembly function into chunks to explore. The first chunk will be between offset **0-55.**

Inspect the registers so you know what you're dealing with:

```
(lldb) po $rdi
```

You'll get `UIView` output which is the `description` method for the `UIView` class. But why is that the first parameter? The function signature seems to indicate it should be a `bool`.

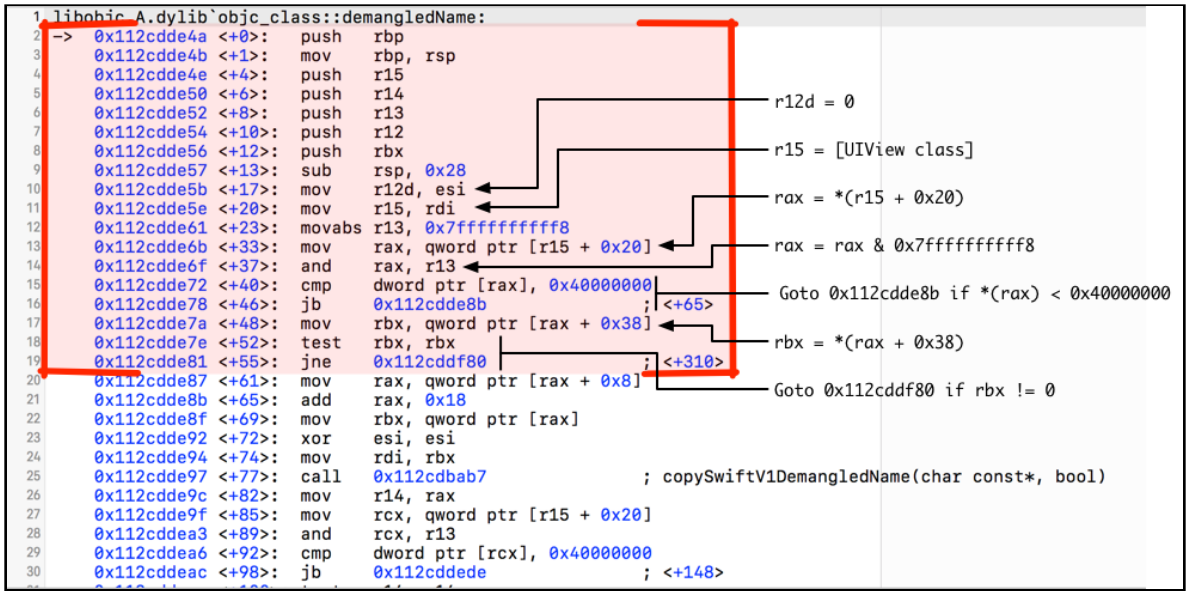
Well, this is a C++ function, and C++ is like Objective-C in the way you call functions on an object. There's an implicit first parameter which is the object the function is being called on. As mentioned throughout this book, the instance passed in as the first register is not always the case with Swift.

Move onto the second param:

```
(lldb) po $rsi
```

You'll get `nil`. This is the `bool` parameter. And `nil` is 0, so this is `false`.

Time to break this thing down. The offsets referred to here are the values within the angle brackets. So offset 13 is `<+13>`.



- **Offset 13**: After this line, the function prologue is over. Time for the actual meat of this function.
- **Offset 17**: This assigns `esi` to `r12d`. This is the Boolean that is passed in. We explored `rsi` earlier and saw it was 0, so `r12d` will be 0 as well.
- **Offset 20**: `rdi` contains the `UIView` class reference and is assigning this value to `r15`.
- **Offset 33**: This offsets `r15` by a value of `0x20` and dereferences it. i.e. `rax = (*([UIView class] + 0x20))`.
- **Offset 37**: The value stored in `rax` is AND'd with `0x7fffffffff8` and stored into `rax`.
- **Offset 48**: The value at `rax` is offset by 0x38 and then dereferenced and stored into `rbx` i.e `rbx = *(rax + 0x38)`.
- **Offset 52-55**: `rbx` is checked for zero. If it returns a non-zero number, then finish up this function, which jumps to `<+310>`, which is right before the function epilogue.

If this check at offset `55` fails (i.e. if `rbx` is 0), execution will continue on to the next assembly instruction, `<+61>`.

The logic between offsets `0–55` is responsible for returning an Objective-C's class back to you as a `char*` if (and only if) that class has been properly loaded. This typically happens when *at least* one method from that class (i.e., that method must be implemented or overriden in that class) is executed.

For example, if a brand new class is called that hasn't created any initializations during the lifetime of your process, the logic between offsets `0–55` will return `nil`. You'll build a **command regex** to confirm this in a second…

Looking at the assembly, you can deduce the following.

If you have an already-initialized class at instance X, and if you offset X by 0x20 and dereference this, the output would look like:

```
*(uint64_t *)(X + 0x20)
```

You then bitwise AND this value with `0x7ffffffffff8`:

```
*(uint64_t *)(X + 0x20) & 0x7ffffffffff8
```

Next, take this value, offset it by 0x38 and dereference that:

```
*(uint64_t *)((*(uint64_t *)(X + 0x20) & 0x7ffffffffff8) + 0x38)
```

This is the final address, so you just need to cast it into the correct type, a `char *`:

```
(char *)*(uint64_t *)((*(uint64_t *)(X + 0x20) & 0x7ffffffffff8) + 0x38)
```

Now, if you have a reference to an `NSObject`, you know from Chapter 25, "Script Bridging with SBValue & Language Contexts" that the memory address right at the start of the object will point to the class itself (the `isa` pointer). If you don't understand that, go back and reread Chapter 25 — or else the remainder of this chapter will get pretty intense. :]

Putting it all together, to get an instance's class name as a `char*`, behold this monstrosity:

```
(char *)*(uint64_t *)((*(uint64_t *)((*(uint64_t *)Instance_of_X) + 0x20)
& 0x7ffffffffff8) + 0x38)
```

Yep, you get to manually replicate this in LLDB to make sure this works!

> **Note**: I'll repeat this once more: this will NOT work for Objective-C classes that haven't been initialized yet. There's a reason why you're using a `UIView`, because if you can see the UI on your screen, then the `UIView` class has definitely been initialized and at least one `UIView` has been created.

In LLDB, go after the `UIView` class:

```
(lldb) p/x [UIView class]
(Class) $1 = 0x000000010c09ce60 UIView
```

You'll get a different address. Copy that to your clipboard.

Take that address and offset it by `0x20` and view the memory at that location:

```
(lldb) x/gx '0x000000010c09ce60 + 0x20'
```

You'll get some value:

```
0x10c09ce80: 0x0000608000064b80
```

`AND` that value with 0x7fffffffff8 (that's 10 f's):

```
(lldb) p/x 0x7fffffffff8 & 0x0000608000064b80
```

You'll get another number:

```
0x0000608000064b80
```

Take that value, offset it by `0x38` and dereference it.

```
(lldb) x/gx '0x0000608000064b80 + 0x38'
```

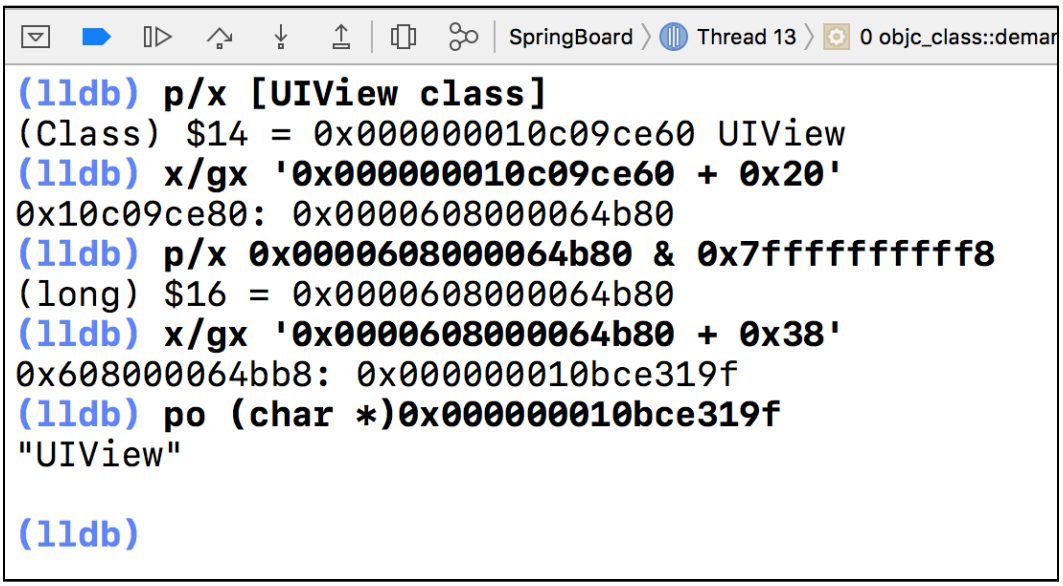
You'll get something like:

```
0x608000064bb8: 0x000000010bce319f
```

See if the value at `0x000000010bce319f` (or at least for me) contains the `char*` pointer.

```
(lldb) po (char *)0x000000010bce319f
```

If everything went well, you'll get your `char*` representation for `UIView`.



Yay! Pointers!

Create a new **regex command** to verify everything I've told you is true.

Just enter this into the console; no need to put this in your `~/.lldbinit` file:

```
command regex getcls 's/(.+)/expression -lobjc -O -- (char *)*(uint64_t
*)((*(uint64_t *)((*(uint64_t *)%1) + 0x20) & 0x7ffffffffff8) + 0x38)/'
```

This grabs the `char*` class name from any instance whose class has already been loaded into your process.

Once you've entered this into your LLDB console, give it a go on the known-to-work `UIView`:

```
(lldb) getcls [UIView new]
```

Now go after something that hasn't been initialized or has had any methods executed for that class, like **`UIAlertController`**:

```
(lldb) getcls [UIAlertController new]
```

You'll get `nil`, since this class hasn't executed any code yet that's unique for the class.

```
(lldb) po [UIAlertController class]
```

Re-execute the `getcls` command:

```
(lldb) getcls [UIAlertController new]
```

You'll now get a reference to the `char*` representation of `UIAlertController`. Remember if any unique method for that class is executed, the Objective-C runtime loads that class in.

Now, the `class` (i.e. `-[NSObject class]`) method is not unique for `UIAlertController`, but guess what is?

You're `po`'ing this object and the `debugDescription` and `description` methods are unique (overridden) to this class.

Therefore, just by `po`'ing a `UIAlertController` class, it'll load it into the runtime!

Run your custom command, `methods`, that you created in Chapter 15, "Dynamic Frameworks" on `UIAlertController` to verify the overriden `debugDescription` method if you have any doubts.

Scary assembly, part II

It's time to revisit the second part of interest in the `objc_class::demangledName(bool)` C++ function. This assembly chunk will focus on what the logic does if the initial location for that `char*` is not in the initial location of interest — that is, if the class isn't loaded yet.

You need to create a breakpoint on assembly instruction offset 61, the instruction immediately following the instruction on offset 55.

You could blindly call classes to see what classes aren't loaded in the runtime, but I haven't a clue what's in your process, and you have no clue what's in mine!

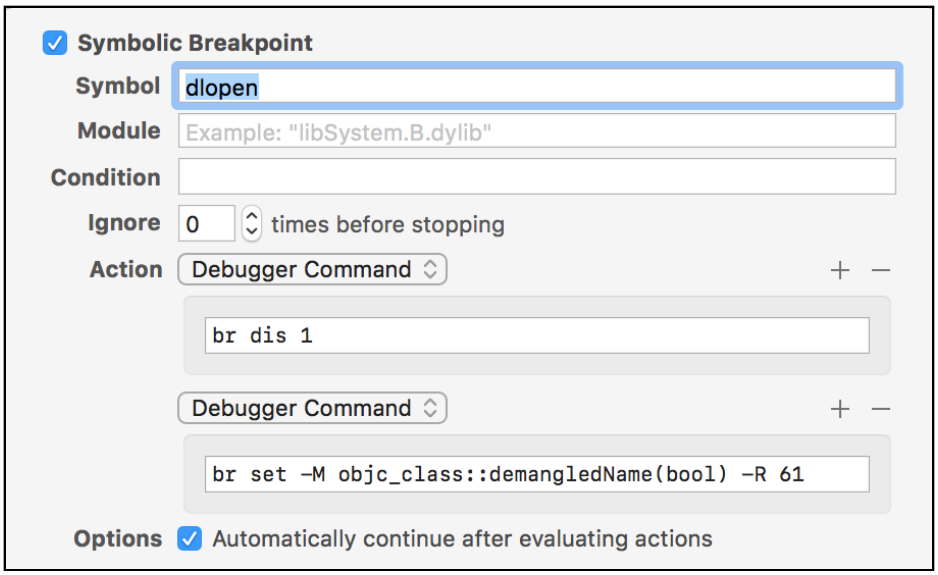
Instead, create a symbolic breakpoint that stops on offset 61 of `objc_class::demangledName(bool)`.

Create a symbolic breakpoint in Xcode using the following details:

- Use **`dlopen`** for the symbol.
- In action 1: remove this breakpoint using **`br dis 1`**.
- In action 2: set a breakpoint on offset 61 of **`objc_class::demangledName(bool)`** with this command:

```
br set -M objc_class::demangledName(bool) -R 61
```

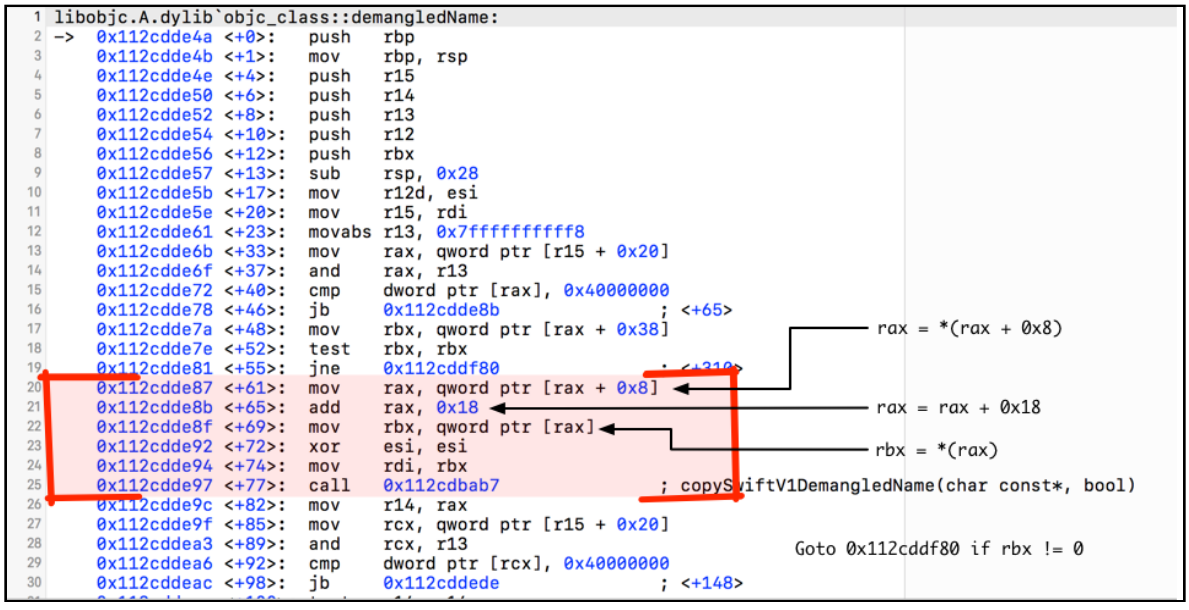
- Select "Automatically continue after evaluating actions".



Rebuild and run the **VCTransitions** application.

You won't get very far into your program before this breakpoint is hit; you can see `dyld` is still busy setting stuff up.

Round two; here we go:



- **Offset 61**: Provided the initial location in memory was `nil`, control continues to 61 where `rax + 0x8` is dereferenced and stored into `rax` again.

- **Offset 65**: The value 0x18 is added to `rax` and stored back into `rax`. `rax` could be a struct that is holding a value of interest, which could explain offsetting this address.
- **Offset 69**: The value at `rax` is dereferenced and stored into `rbx`, which will get passed into `rdi` 2 instructions later. After that, a call instruction occurs, which by the disassembly commentary, looks to expect a `char const *` as the first parameter.

This is the "interesting" part of this function to you. After that, this function calls the **`copySwiftV1DemangledName`** function and sets up the logic to load the class into the Objective-C runtime.

But for you, this is as far as you need to explore this function.

Feel free to ensure that `rdi` will always produce a valid `char*` at offset 77, but again, that will be something you can do on your own time. You've still got a DTrace script to write.

Converting research into code

You've done the necessary research to figure out how to traverse memory to get the character array representation of a class. Time to implement this thing.

Included in the **starter** script is a skeleton DTrace script named **msgsendsnoop.d**.

You'll start with this DTrace script and build out the code for it. Once working and tested, you'll transfer that code into a LLDB Python script, which will dynamically generate the code you want.

In Terminal **`cd`** into the **starter** directory. You can drag and drop the directory into Terminal to autocomplete.

`cat` the contents of this script:

```
cat ./msgsendsnoop.d
```

Here's the output from `cat`:

```
#!/usr/sbin/dtrace -s
#pragma D option quiet

dtrace:::BEGIN
{
    printf("Starting... Hit Ctrl-C to end.\n");
}

pid$target::objc_msgSend:entry
{
  this->selector = copyinstr(arg1);
```

```
    printf("0x%016p, +|-[%s %s]\n", arg0, "__TODO__",
                                          this->selector);
}
```

Let's break this down. This script will stop on the **`objc_msgSend entry probe`** with the appropriate PID passed in (thanks to the pid$target `provider`). Once hit, the selector's `char*` is copied into the kernel and printed.

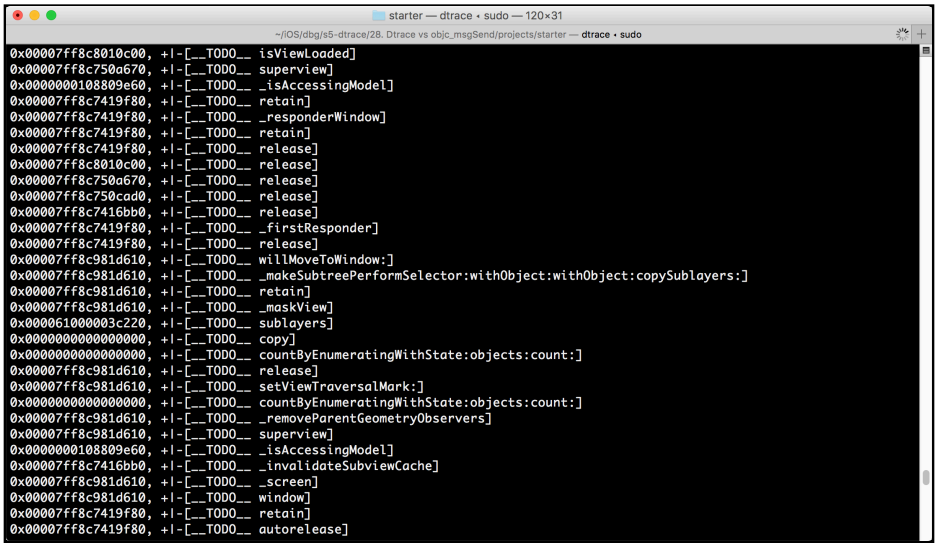
As an example of what will happen, let's say a `-[UIView initWithFrame:]` is about to be called. The following will print out:

```
0x00000000deadbeef, +|-[__TODO__ initWithFrame:]
```

Verify this is true by tracing all the `objc_msgSend` calls in the **VCTransitions** application:

```
sudo ./msgsendsnoop.d -p `pgrep VCTransitions`
```

Tap around on some classes. Hopefully this gives you an idea of how frequently this method gets called.



It's time to fix that annoying `__TODO__` and replace it with the actual name of the class.

Open up **msgsendsnoop.d** and replace the existing `pid$target::objc_msgSend:entry` code with the following:

```
pid$target::objc_msgSend:entry
{
  /* 1 */
  this->selector = copyinstr(arg1);
  /* 2 */
  size = sizeof(uintptr_t);
  /* 3 */
```

```
    this->isa = *((uintptr_t *)copyin(arg0, size));

    /* 4 */
    this->rax = *((uintptr_t *)copyin((this->isa + 0x20), size));
    this->rax =  (this->rax & 0x7fffffffff8);

    /* 5 */
    this->rbx = *((uintptr_t *)copyin((this->rax + 0x38), size));

    this->rax = *((uintptr_t *)copyin((this->rax + 0x8),  size));

    /* 6 */
    this->rax = *((uintptr_t *)copyin((this->rax + 0x18), size));

    /* 7 */
    this->classname = copyinstr(this->rbx != 0 ?
                                this->rbx  : this->rax);
    printf("0x%016p +|-[%s %s]\n", arg0, this->classname,
                                         this->selector);
}
```

Note: I would recommend you type in each line and make sure it runs, instead of typing in everything at once. Some DTrace script errors can be tricky to hunt down.

Deep breath. Here's what each line does:

1. **`this->selector`** does a `copyinstr`, because you know the second parameter (aka `arg1`) is an Objective-C selector (aka a C string). Since C `char*`s end with a null character, DTrace can automatically determine how much data to read.

2. In a moment, you're going to `copyin` some data. However, `copyin` expects a size, because unlike a string, DTrace doesn't know when the arbitrary data ends. You declare a variable named **`size`**, which equals the length of a pointer. In x64, this will be 8 bytes.

3. This is getting the reference to the class of the instance. Remember, the dereferenced pointer at the start address of a Objective-C or Swift instance will point to the class.

4. Now for the fun part you learned about from the assembly in **`objc_class::demangledName(bool)`**. You'll replicate the logic found in the registers, as well as even use the same names for the registers! You're using `rax` to mimic the logic that this function performs.

5. This is the logic where `(rax + 0x38)` gets set to `this->rbx`, just like in the actual assembly.

6. This is the final line if the value `this->rbx` is 0 (aka the class has not been loaded yet).
7. You are using a ternary operator to figure out which clause local variable to use. If `this->rbx` is non-null, use it. Otherwise, reference `this->rax`.

Save your work. Jump over to Terminal and relaunch this DTrace script:

```
sudo ./msgsendsnoop.d -p `pgrep VCTransitions`
```

Wooooooooooooooooot! That crazy hack actually worked!

Scanning the content in your script, it looks like the script is throwing errors occasionally when `objc_msgSend` is calling a `nil` object (i.e. `RDI`, aka `arg0`, is 0x0).

You can view only the errors with the following command:

```
sudo ./msgsendsnoop.d -p `pgrep VCTransitions` | grep invalid
```

Let's fix that now with a simple predicate. Immediately following `pid$target::objc_msgSend:entry`, add the following predicate so it looks like this:

```
pid$target::objc_msgSend:entry / arg0 > 0x100000000 /
```

This says, "Don't run this DTrace action if the first param is `nil` or a section of memory that is not utilized."

Typically, in a macOS userland process, this section of memory is off-limits for reading, writing, and executing. If anything is below the number `0x100000000`, DTrace ain't gonna like it, along with anything else reading memory there.

Therefore, if it's below that number, just have DTrace skip it. You can of course, confirm this using LLDB with the following command:

```
(lldb) image dump sections VCTransitions
```

But that's for you to verify when you're bored. You still gotta finish this script.

Removing noise

To be honest, I couldn't care less about tracing memory-management code the compiler has generated. This means anything with **`retain`** or **`release`** needs to get outta here.

Make a new clause with the same DTrace probe above your current probe:

```
pid$target::objc_msgSend:entry
{
  this->selector = copyinstr(arg1);
```

```
}

/* old code below */
pid$target::objc_msgSend:entry / arg0 > 0x100000000 /
```

You're now declaring the selector in a new clause before the main clause with all the memory jumping logic. This will let you filter Objective-C methods inside the predicate section of the main clause.

Speaking of which, augment the predicate in the main clause now:

```
pid$target::objc_msgSend:entry / arg0 > 0x100000000 / &&
                    this->selector != "retain" &&
                    this->selector != "release" /
```

This will now ignore any Objective-C selectors that equal `retain` or `release`.

While you're at it, there's no need to reassign the `this->selector` in the main clause now you're doing it in the other one. Although it isn't harmful, it's superfluous logic. Remove it, or don't... whatever makes you happy.

Your two clauses should now (hopefully somewhat?) look like this:

```
pid$target::objc_msgSend:entry
{
  this->selector = copyinstr(arg1);
}

pid$target::objc_msgSend:entry / arg0 > 0x100000000 &&
                    this->selector != "retain" &&
                    this->selector != "release" /
{
  size = sizeof(uintptr_t);
  this->isa = *((uintptr_t *)copyin(arg0, size));

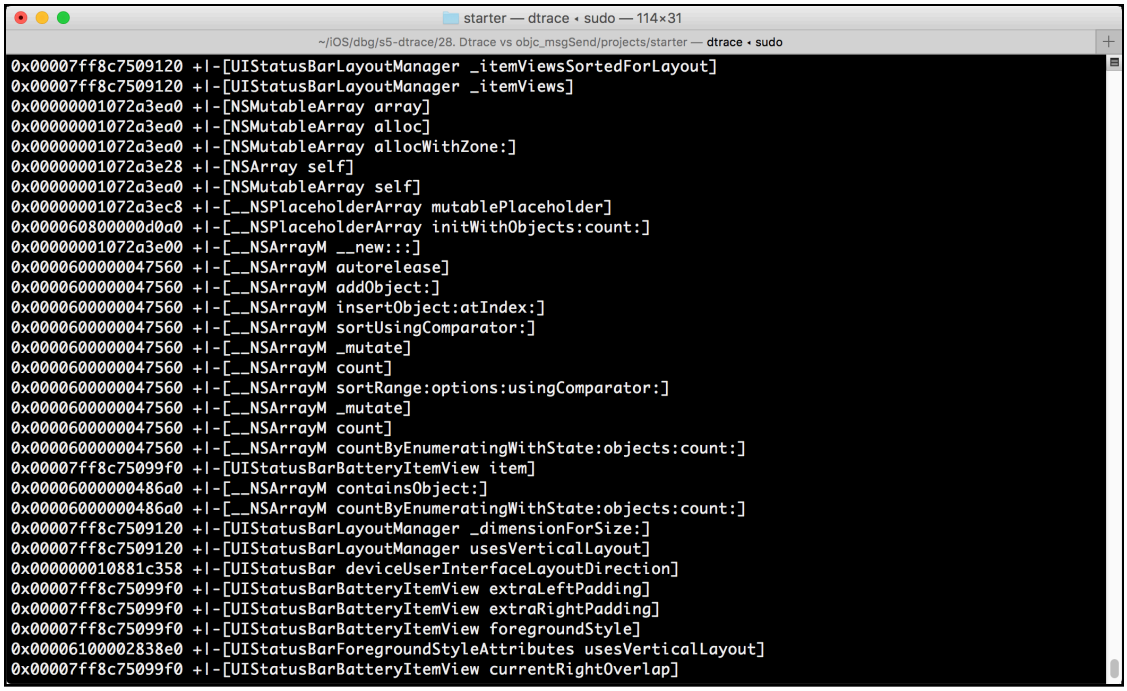
  this->rax = *((uintptr_t *)copyin((this->isa + 0x20), size));
  this->rax =  (this->rax & 0x7ffffffffff8);
  this->rbx = *((uintptr_t *)copyin((this->rax + 0x38), size));

  this->rax = *((uintptr_t *)copyin((this->rax + 0x8),  size));
  this->rax = *((uintptr_t *)copyin((this->rax + 0x18), size));

  this->classname = copyinstr(this->rbx != 0 ?
                              this->rbx  : this->rax);
  printf("0x%016p +|-[%s %s]\n", arg0, this->classname,
                                       this->selector);
}
```

Relaunch the script:

```
sudo ./msgsendsnoop.d -p `pgrep VCTransitions`
```



Oh man, that's sooo much better.

But still, that's too much noise. Time to take this script and combine it with LLDB to only produce output that pertains to code in the main executable.

Limiting scope with LLDB

Included within the **starter** folder is a LLDB Python script that creates a DTrace script and runs it with the *exact* logic you've just implemented.

Womp womp… spoiler alert. You could have just used that script in the first place. But that wouldn't have been as much fun.

This file is named **snoopie.py**. Take this file and copy it into your `~/lldb` directory. If you've followed along with Chapter 26, "SB Examples, Improved Lookup", you have a script in there named `lldbinit.py` that automatically loads all the scripts in the same directory for you.

If you were too cool for school and didn't do that chapter, you'll need to add the following line of code into your `~/.lldbinit` file:

```
command script import ~/lldb/snoopie.py
```

You'll use a creative solution to filter out the code in this DTrace script to only trace Objective-C/dynamic Swift code belonging to the VCTransitions executable. Normally, when snooping code in a framework, I'll often grab the `__TEXT` segment of a module and compare the instruction pointer to the upper and lower bounds of the `__TEXT` segment that's loaded in memory (the area in memory containing executable code). If the instruction pointer is between the upper and lower bounds, then you can assume you want to use DTrace to trace the code.

Unfortunately, you're going after `objc_msgSend`, the chokepoint used for Objective-C code in all modules. This means that you can't rely on the instruction pointer to tell you which module you're in.

Instead, you need to go about this by isolating the addresses of a class to only be contained within the **`__DATA`** segment of the main executable.

Head on back to your Xcode project, **VCTransitions**.

Build, run, stop execution and bring up LLDB. Then type the following:

```
(lldb) p/x (void *)NSClassFromString(@"ObjCViewController")
```

You'll get the address to the `ObjCViewController` class:

```
(void *) $0 = 0x000000010db34080
```

Take this address and determine what section of memory this thing is located in.

```
(lldb) image lookup -a 0x000000010db34080
```

You'll get something similar to the following:

```
Address: VCTransitions[0x0000000100012080]
(VCTransitions.__DATA.__objc_data + 40)
Summary: (void *)0x000000010db34058
```

Therefore, you can deduce this class is within the **VCTransitions `__DATA`** segment inside the **`__objc_data`** section. You'll use the LLDB Python module to find the upper and lower bounds of this `__DATA` segment.

Now you're going to use the good old `script` command to find how you can create this code through the LLDB module. Back in LLDB, type the following:

```
(lldb) script path = lldb.target.executable.fullpath
```

This will give you the **`SBFileSpec`** representing the executable, **VCTransitions**, and assign it to the variable **`path`**. Print out the path to make sure it's valid:

```
(lldb) script path
```

You'll get the full path to the location of this executable. You can use this `path` variable to get the correct **`SBModule`** from the **`SBTarget`**. Type the following into LLDB:

```
(lldb) script print lldb.target.module[path]
```

You'll get the `SBModule` representing the main executable.

Within a `SBModule`, there's **SBSection**s. You can get all sections within an `SBModule` using the **sections** property, or you can get a specific section using **section[index]**. Yep, that property conforms to Python's `__getitem__`. Type the following into LLDB:

```
(lldb) script print lldb.target.module[path].section[0]
```

You'll get something like:

```
[0x0000000000000000-0x0000000100000000) VCTransitions.__PAGEZERO
```

The implementation of `__getitem__` can also let `SBSection` act as a dictionary. So you can also access the `__PAGEZERO` section like so:

```
(lldb) script print lldb.target.module[path].section['__PAGEZERO']
```

This means you can easily access the `__DATA` `SBSection` by using the following:

```
(lldb) script print lldb.target.module[path].section['__DATA']
```

Cool, that works. Assign this `SBSection` to a variable named **`section`**, like so:

```
(lldb) script section = lldb.target.module[path].section['__DATA']
```

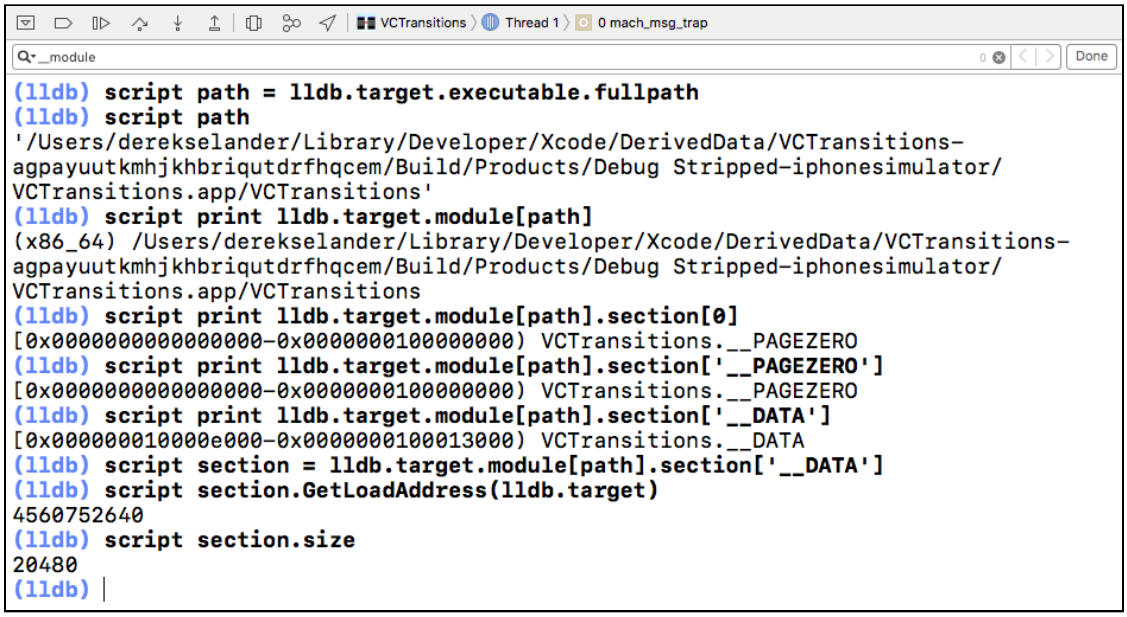
You now have a reference to the correct segment. There are segments in the __DATA section you can dissect, but you might as well grab the whole section, since it's one contiguous region in memory.

Get the **`load address`** for the section, like so:

```
(lldb) script section.GetLoadAddress(lldb.target)
```

This will print the start location. Grab the size as well, while you're at it:

```
(lldb) script section.size
```



So what does this give you? You can make a DTrace predicate that checks if the class is in between these values in memory. If they are, execute the DTrace action. If they're not, ignore. Let's implement this!

Fixing up the snoopie script

As indicated, this **snoopie.py** script works as-is, so you're just going to add some small logic to the predicate to filter only instances.

Open up **~/lldb/snoopie.py** and navigate to the **`generateDTraceScript`** function. Remove the `dataSectionFilter = ...` line.

Then add the following code in its place:

```
target = debugger.GetSelectedTarget()
path = target.executable.fullpath
section = target.module[path].section['__DATA']
start_address = section.GetLoadAddress(target)
end_address = start_address + section.size

dataSectionFilter = '''{} <= *((uintptr_t *)copyin(arg0,
    sizeof(uintptr_t))) &&
   *((uintptr_t *)copyin(arg0, sizeof(uintptr_t))) <= {}
'''.format(start_address, end_address)
```

The interesting point here is you're taking the `arg0` and dereferencing it if (and only if) `arg0` is greater than `0x100000000`, which indicates a valid instance in memory.

That's it! No more code! You're all done!

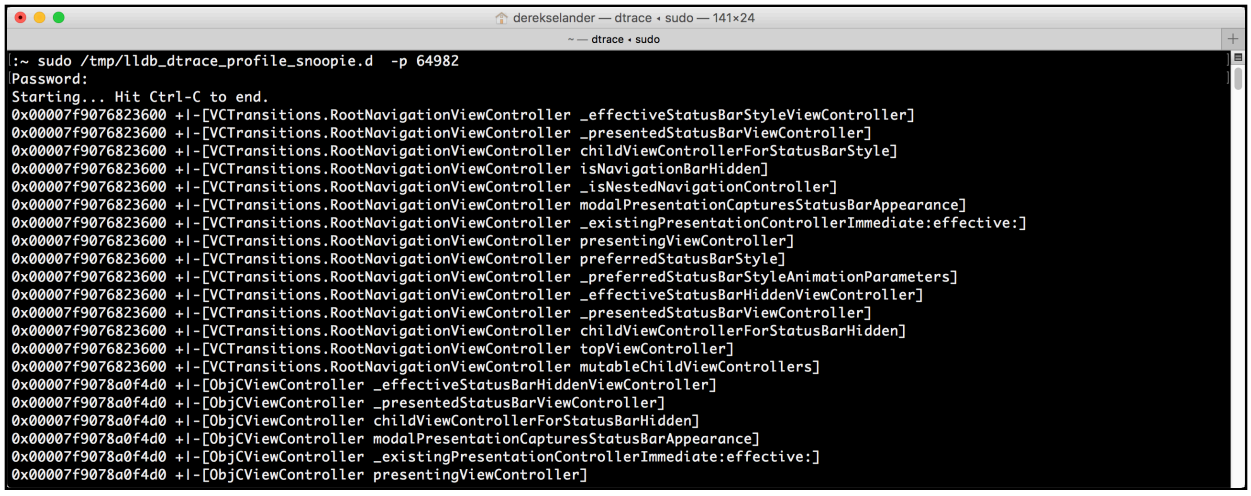
Save your work, jump over to the LLDB console, reload the contents in LLDB either through your custom `reload_script` command or manually by command `script import ~/.lldbinit`.

Once reloaded, try this thing out. In LLDB:

```
(lldb) snoopie
```

Paste the contents to a Terminal window and have fun.

DTrace now only profiles code that's in your main (stripped) executable.



Have fun with this script on some other apps on your computer!

Where to go from here?

You've got some homework to do on your end. This script will not play nicely with Objective-C categories. For example, there could be a class that's implemented within a different module, which has an Objective-C category implemented within the main executable. You'll need to figure out some creative way to check if the Objective-C selector in `objc_msgSend` was implemented within the main executable or not.

In addition, the `printf` in your current code doesn't indicate whether `arg0` is a class method or not. You'll need to figure out how to determine if the `arg0` parameter is a class or an instance solely by jumping through memory.

How can you go about finding this?

- If `arg0` is an instance of a class, the `isa` param will point to a non-meta class.
- If `arg0` is the class, then the `isa` param will point to the meta class.
- Explore the assembly of **`class_isMetaClass`** to determine what values inside a `Class` indicate if it's a meta class or not.

Once you've found how to jump through memory to determine if a class is a meta class or not, replicate the logic found in **`class_isMetaClass`** in your DTrace script.

Since this is either an instance of a class or the `Class` object itself, you can use a **ternary** operator inside your DTrace script with something similar to the following:

```
this->isMeta = ... // logic here
this->isMetaChar = this->isMeta ? '+' : '-'

printf("0x%016p %c[%s %s]\n", arg0, this->isMetaChar,
                                    this->classname,
                                    this->selector);
```

Heh… `isMetaChar`. That will totally be a Pokémon name some day.

Good luck!

Appendix A: LLDB Cheatsheet

A cheatsheet for commands and ideas on how to use LLDB.

Getting help

```
(lldb) help
```

List all commands and aliases.

```
(lldb) help po
```

Get help documentation for `po` (`expression`) command.

```
(lldb) help break set
```

Get help documentation for `breakpoint set`.

```
(lldb) apropos step-in
```

Search through help documentation containing `step-in`.

Finding code

```
(lldb) image lookup -rn UIAlertController
```

Look up all code containing `UIAlertController` that's compiled or loaded into an executable.

```
(lldb) image lookup -rn (?i)hosturl
```

Case insensitive search for any code that contains `"hosturl"`.

```
(lldb) image lookup -rn 'UIViewController\ set\w+:\]'
```

Look up all setter property methods `UIViewController` implements or overrides.

```
(lldb) image lookup -rn . Security
```

Look up all code located within the `Security` module.

```
(lldb) image lookup -a 0x10518a720
```

Look up code based upon address `0x10518a720`.

```
(lldb) image lookup -s mmap
```

Look up code for the symbol named `mmap`.

Breakpoints

```
(lldb) b viewDidLoad
```

Creates a breakpoint on all methods named `viewDidLoad` for both Swift and Objective-C.

```
(lldb) b setAlpha:
```

Creates a breakpoint on either the `setAlpha:` Objective-C method or the setter of the Objective-C `alpha` property.

```
(lldb) b -[CustomViewControllerSubclass viewDidLoad]
```

Creates a breakpoint on the Objective-C method `[CustomViewControllerSubclass viewDidLoad]`.

```
(lldb) rbreak CustomViewControllerSubclass.viewDidLoad
```

Creates a regex breakpoint to match either an Objective-C or Swift class `CustomViewControllerSubclass` which contains `viewDidLoad`. Could be Objective-C `-[CustomViewControllerSubclass viewDidLoad]` or could be Swift `ModuleName.CustomViewControllerSubclass.viewDidLoad () -> ()`.

```
(lldb) breakpoint delete
```

Deletes all breakpoints.

```
(lldb) breakpoint delete 2
```

Deletes breakpoint ID 2.

```
(lldb) breakpoint list
```

List all breakpoints and their IDs.

```
(lldb) rbreak viewDid
```

Creates a regex breakpoint on `.*viewDid.*`.

```
(lldb) rbreak viewDid -s SwiftRadio
```

Creates a breakpoint on `.*viewDid.*`, but restricts the breakpoint(s) to the `SwiftRadio` module.

```
(lldb) rbreak viewDid(Appear|Disappear) -s SwiftHN
```

Creates a breakpoint on `viewDidAppear` or `viewDidDisappear` inside the `SwiftHN` module.

```
(lldb) rb "\-\[UIViewController\ set" -s UIKit
```

Creates a breakpoint on any Objective-C style breakpoints containing `-[UIViewController set` within the UIKit module.

```
(lldb) rb . -s SwiftHN -o 1
```

Create a breakpoint on every function in the `SwiftHN` module, but remove all breakpoints once the breakpoint is hit.

```
(lldb) rb . -f ViewController.m
```

Create a breakpoint on every function found in `ViewController.m`.

Expressions

```
(lldb) po "hello, debugger"
```

Prints `"hello, debugger"` regardless of the debugging context.

```
(lldb) expression -lobjc -O -- [UIApplication sharedApplication]
```

Print the shared `UIApplication` instance in an Objective-C context.

```
(lldb) expression -lswift -O -- UIApplication.shared
```

Print the shared `UIApplication` instance in a Swift context.

```
(lldb) b getenv
(lldb) expression -i0 -- getenv("HOME")
```

Creates a breakpoint on `getenv`, executes the `getenv` function, and stops at the beginning of the `getenv` function.

```
(lldb) expression -u0 -O -- [UIApplication test]
```

Don't let LLDB unwind the stack if you're executing a method that will cause the program to crash.

```
(lldb) expression -p -- NSString *globalString = [NSString
stringWithUTF8String: "Hello, Debugger"];
(lldb) po globalString
Hello, Debugger
```

Declares a global `NSString*` called `globalString`.

```
(lldb) expression -g -O -lobjc -- [NSObject new]
```

Debug the debugger that's parsing the `[NSObject new]` Objective-C expression.

Stepping

```
(lldb) thread return false
```

Return early from code with `false`.

```
(lldb) thread step-in
(lldb) s
```

Step in.

```
(lldb) thread step-over
(lldb) n
```

Step over.

```
(lldb) thread step-out
(lldb) finish
```

Step out of a function.

```
(lldb) thread step-inst
(lldb) ni
```

Step in if about to execute a function. Step an assembly instruction otherwise.

GDB formatting

```
(lldb) p/x 128
```

Print value in hexadecimal.

```
(lldb) p/d 128
```

Print value in decimal.

```
(lldb) p/t 128
```

Print value in binary.

```
(lldb) p/a 128
```

Print value as address.

```
(lldb) x/gx 0x00000010fff6c40
```

Get the value pointed at by `0x00000010fff6c40` and display in 8 bytes.

```
(lldb) x/wx 0x00000010fff6c40
```

Get the value pointed at by `0x00000010fff6c40` and display in 4 bytes.

Memory

```
(lldb) memory read 0x00000010fff6c40
```

Read memory at address `0x00000010fff6c40`.

```
(lldb) po id $d = [NSData dataWithContentsOfFile:@"..."]
(lldb) mem read `(uintptr_t)[$d bytes]` `(uintptr_t)[$d bytes] +
(uintptr_t)[$d length]` -r -b -o /tmp/file
```

Grab an instance of a remote file and write it to `/tmp/file` on your computer.

Registers & assembly

```
(lldb) register read -a
```

Display all registers on the system.

```
(lldb) register read rdi rsi
```

Read the `RSI` and the `RDI` register in x64 assembly.

```
(lldb) register write rsi 0x0
```

Set the `RSI` register to 0x0 in x64 assembly.

```
(lldb) register write rflags `$rflags ^ 64`
```

Toggle the zero flag in x64 assembly (augment if condition logic).

```
(lldb) register write rflags `$rflags | 64`
```

Set the zero flag (set to 1) in x64 assembly (augment if condition logic).

```
(lldb) register write rflags `$rflags & ~64`
```

Clear the zero flag (set to 0) in x64 assembly (augment if condition logic).

```
(lldb) register write pc `$pc+4`
```

Increments the program counter by 4.

```
(lldb) disassemble
```

Display assembly for function in which you're currently stopped.

```
(lldb) disassemble -p
```

Disassemble around current location; useful if in the middle of a function.

```
(lldb) disassemble -b
```

Disassemble function while showing opcodes; useful for learning what is responsible for what.

```
(lldb) disassemble -n '-[UIViewController setTitle:]'
```

Disassemble the Objective-C `-[UIViewController setTitle:]` method.

```
(lldb) disassemble -a 0x000000010b8d972d
```

Disassemble the function that contains the address 0x000000010b8d972d.

Modules

```
(lldb) image list
```

List all modules loaded into the executable's process space.

```
(lldb) image list -b
```

Get the names of all the modules loaded into the executable's process space.

```
(lldb) process load /Path/To/Module.framework/Module
```

Load the module located at path into the executable's process space.

Appendix B: Python Environment Setup

It's not my place to force an IDE on you for Python development. However, if you're actively looking for a Python editor for the Python related chapters — then we should have a little chat.

Getting Python

Good news: if you have a Mac, it automatically ships (at the time of writing) with Python version 2.7. This is the same version LLDB uses.

If, for some weird reason, you like to `rm` random things in Terminal and you need to reinstall Python, you can download Python here: https://www.python.org/downloads/. Make sure to download the version of Python that matches the version packaged with LLDB. If you're not sure which version to get, you can get the LLDB Python version through Terminal:

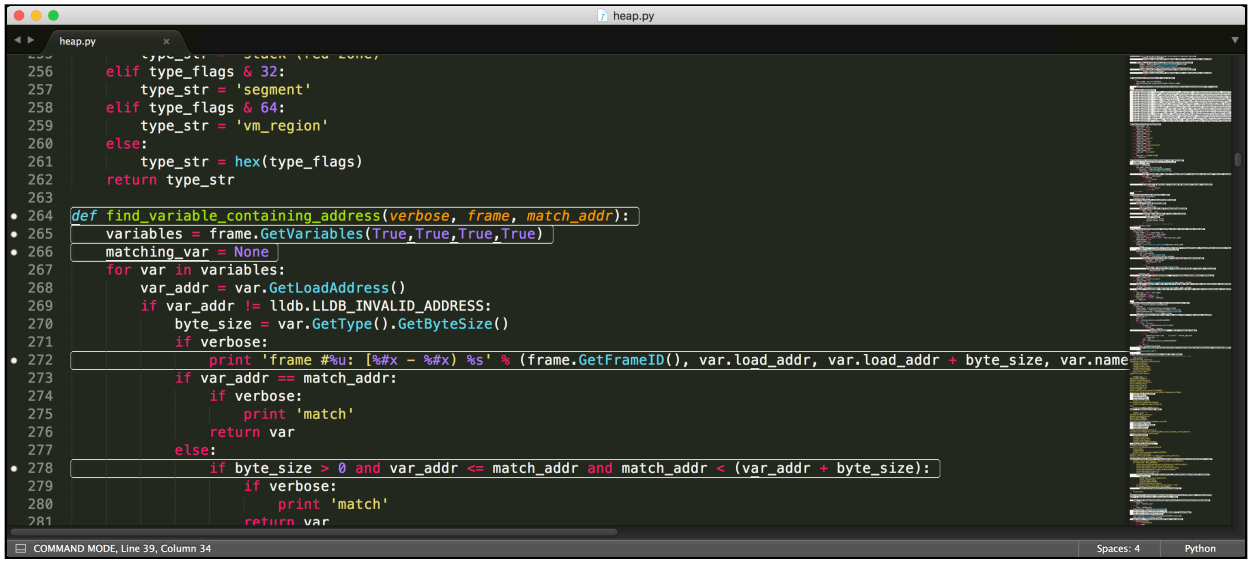
```
lldb
(lldb) script import sys; print sys.version
```

Don't worry about the final part of the version number. If you have 2.7.12 and LLDB quotes 2.7.10, that will work just fine.

Python text editors

A list of Python editors can be found here: https://wiki.python.org/moin/PythonEditors.

For the small, quick Python scripts you'll write in this book, I would recommend using **Sublime Text**. Sublime Text can be found at https://www.sublimetext.com/; although it's a paid application, it's free to try with no time limit.



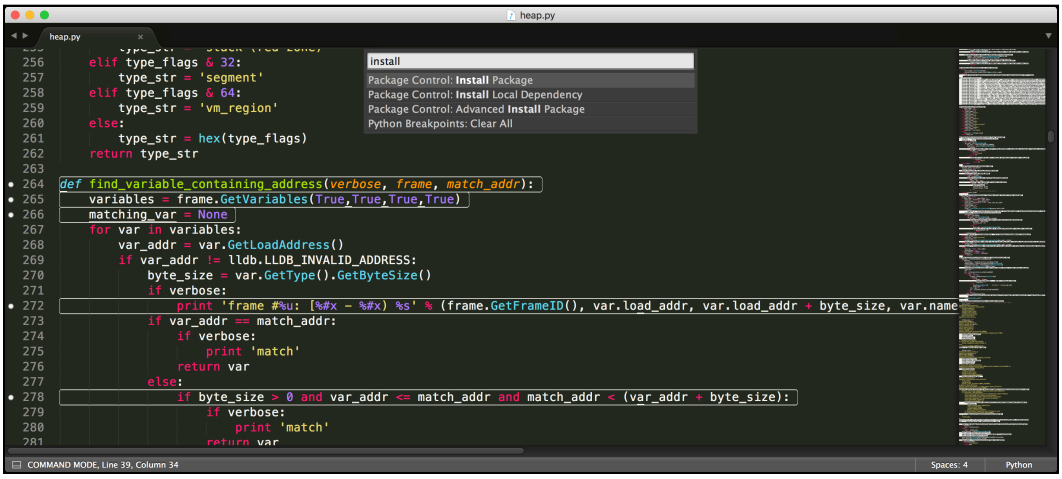
Both this book, as well as all my LLDB Python scripts, were written (and debugged) through Sublime Text 3.

You'll likely want to install a couple of additional components to Sublime Text to make developing and debugging LLDB Python scripts easier.

The easiest way to install these additional components is to use the Sublime Text Package Control, which is an excellent package manager for Sublime Text. You can find instructions on how to install the Package Control at https://packagecontrol.io/installation.

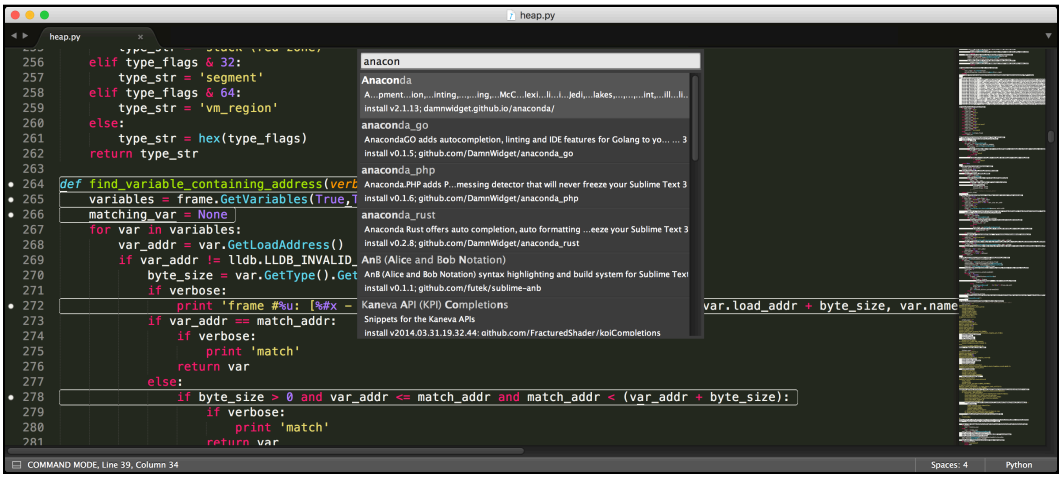
Once installed, you'll be able to easily search for new components designed for Sublime Text by pressing ⌘ + **Shift** + **P** and typing **install**. A selection item of **Package Control: Install Package** will appear.

Select this option:



After the package manager has been installed, you can search for packages that will help you in Python development. Here are a few packages I would recommend using if you're developing in Python:

- **AutoPep8**: Automatically formats Python code to conform to the PEP 8 style guide using `autopep8` and `pep8` modules. https://packagecontrol.io/packages/AutoPEP8
- **PythonBreakpoints**: A Sublime Text plugin to quickly set Python breakpoints by injecting the `set_trace()` call of `pdb` or another debugger of your choice. https://packagecontrol.io/packages/Python%20Breakpoints
- **Anaconda**: Anaconda turns your Sublime Text 3 into a fully featured Python development IDE including autocompletion, code linting, `autopep8` formatting, `McCabe` complexity checker `Vagrant` and `Docker` support for Sublime Text 3 using `Jedi`, `PyFlakes`, `pep8`, `MyPy`, `PyLint`, `pep257` — and `McCabe` will never freeze your Sublime Text 3. https://packagecontrol.io/packages/Anaconda



Working with the LLDB Python module

When working with Python, you'll often import modules to execute code or classes within that module. When working with LLDB's Python module, you'll sometimes come across an `import lldb` somewhere in the script, usually right at the top.

By default, Xcode will launch a version of Python that's bundled within Xcode. When Xcode launches this bundled version of Python, the path to where the lldb module is located is set up automatically. However, in your normal Python development, you won't have access to this module if you were to execute your script through Sublime Text. As a result, you'll need to modify your **`PYTHONPATH`** environment variable to include the appropriate directory where the lldb Python module lives.

In Terminal, ensure your **`~/.bash_profile`** exists:

```
touch ~/.bash_profile
```

Open `.bash_profile` file in your favorite text editor (like Sublime!) and add the following line of code:

```
export PYTHONPATH=/Applications/Xcode.app/Contents/SharedFrameworks/
LLDB.framework/Versions/A/Resources/Python:$PYTHONPATH
```

> **Note**: This assumes your Xcode is located at `/Applications/Xcode.app`. If it isn't, because you particularly like being different, then you'll need to change the path.

Save and close the file. You'll be able to access the lldb module from any Python session on your computer.

Doing this gives you the advantage of checking for syntax errors in Sublime Text (or equivalent) during debugging time — instead of finding a syntax error when your script is loaded into LLDB.

Appendix C: LLDB Bug

In LLDB version `1000.11.37.1` (the version that's packaged with Xcode 10.0), there's a bug in LLDB that incorrectly imports the wrong headers when debugging an iOS Simulator or iOS target. Fortunately, this bug only appears when running in a Terminal session.

To see if you're affected, open up a Terminal window and check your LLDB version:

```
(lldb) version
lldb-1000.11.37.1
  Swift-4.2
```

If you have the same version (or maybe one coming from Xcode 10.1/10.2) this will happen when debugging any iOS Simulator application:

```
(lldb) po @import UIKit
error: while importing modules:
error: Header search couldn't locate module UIKit
```

This is because LLDB is looking in the MacOS SDK directory for the `UIKit` headers. This also means many LLDB scripts which rely on this feature will also fail.

Solution

Fortunately, a solution to this problem can be found here: https://github.com/DerekSelander/lldb_fix

C Conclusion

Wow! You made it all the way to this conclusion! You either must have jumped straight to this page or you're way more masochistic than I could have anticipated.

If you have any questions or comments about the projects or concepts in this book, or have any stories to tell from your own debugging adventures, please stop by our forums at http://forums.raywenderlich.com.

From here, you have a few paths to explore depending on what you found most interesting in this book.

- If exploring code in Python to make better debugging scripts interests you, then you might want to see what other modules exist in Python 2.7 (or the equivalent Python version LLDB has) to see how far down the rabbit hole you can go. You can find the list of modules in Python 2.7 here: https://docs.python.org/2/py-modindex.html or hunt down one of the many books on Amazon about Python.
- If reverse engineering Apple internals interests you, I would strongly recommend you check out **Jonathan Levin**'s work on anything related to Apple, namely his updated books like **MacOS and iOS Internals, Volume III: Security & Insecurity** or **MacOS and iOS Internals, Volume I: User Mode** at http://www.newosxbook.com/.
- Also check out @snakeninny's free book, https://github.com/iosre/iOSAppReverseEngineering/
- If more generic reverse engineering/hacking interests you, then you might be interested in **Hacking: The Art of Exploitation, 2nd Edition** by **Jon Erickson** at https://www.nostarch.com/hacking2.htm.

- If you want the equivalent of an LLDB newsletter, I would recommend to (nicely!) stalk **Jim Ingham**'s activity on Stack Overflow http://stackoverflow.com/users/2465073/jim-ingham. He works on LLDB at Apple, and combing through his responses on StackOverflow will give you a tremendous amount of insight into LLDB. In addition, check out the LLDB archives http://lists.llvm.org/pipermail/lldb-dev/. There's a lot to dig through, but you can find some incredibly useful hidden gems from the LLDB authors.
- If DTrace interested you, check out http://www.brendangregg.com/dtracebook/index.html. This book will cover a much wider range of how to use DTrace than what I've discussed.

And finally... here's a diff of Jake when this book project began in June 2016 to when my editors finally ripped this book from my cold, lifeless fingers.



Yeah, I am totally that annoying dude on Facebook that constantly posts pictures of his children and/or dogs. :]

Thank you for purchasing this book. Your continued support is what makes the books, tutorials, videos and other things we do at raywenderlich.com possible. We truly appreciate it!

– Derek, Darren, Matt and Chris

The *Advanced Apple Debugging & Reverse Engineering* team

Want to Grow Your Skills?

We hope you enjoyed this book! If you're looking for more, we have a whole library of books waiting for you at https://store.raywenderlich.com.

New to iOS or Swift?

Learn how to develop iOS apps in Swift with our classic, beginner editions.

iOS Apprentice



The iOS Apprentice is a series of epic-length tutorials for beginners where you'll learn how to build four complete apps from scratch. Each new app will be a little more advanced than the one before. By the end of the series you'll be experienced enough to turn your ideas into real apps that you can sell on the App Store. These tutorials have easy to follow step-by-step instructions. https://store.raywenderlich.com/products/ios-apprentice

Swift Apprentice

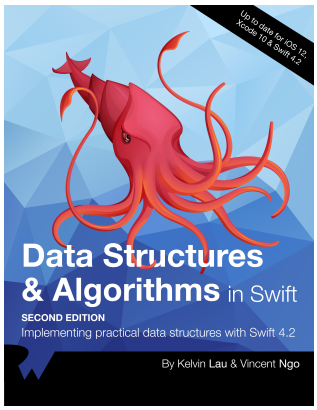


This is a book for complete beginners to Apple's brand new programming language — Swift 4. Everything can be done in a playground, so you can stay focused on the core Swift 4 language concepts like classes, protocols, and generics. This is a sister book to the iOS Apprentice; the iOS Apprentice focuses on making apps, while Swift Apprentice focuses on the Swift 4 language itself. https://store.raywenderlich.com/products/swift-apprentice

Experienced iOS developer?

Level up your development skills with a deep dive into our many intermediate to advanced editions.

Data Structures and Algorithms in Swift



In Data Structures and Algorithms in Swift, you'll learn the most popular and useful data structures, and when and why you should use one particular datastructure or algorithm over another. This set of basic data structures and algorithms will serve as an excellent foundation for building more complex and special-purpose constructs. https://store.raywenderlich.com/products/data-structures-and-algorithms-in-swift

Realm: Building Modern Swift Apps with Realm Database



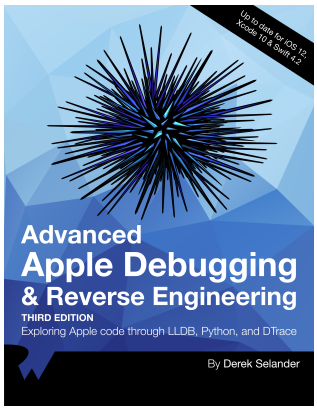
Realm Platform is a relatively new commercial product which allows developers to automatically synchronize data not only across Apple devices but also between any combination of Android, iPhone, Windows, or macOS apps. In this book, you'll take a deep dive into the Realm Database, learn how to set up your first Realm database, see how to persist and read data, find out how to perform migrations and more. https://store.raywenderlich.com/products/realm-building-modern-swift-apps-with-realm-database

Server Side Swift with Vapor



If you're a beginner to web development, but have worked with Swift for some time, you'll find it's easy to create robust, fully featured web apps and web APIs with Vapor 3. This book starts with the basics of web development and introduces the basics of Vapor; it then walks you through creating APIs and web backends; creating and configuring databases; deploying to Heroku, AWS, or Docker; testing your creations and more. https://store.raywenderlich.com/products/server-side-swift-with-vapor

Apple Debugging and Reverse Engineering



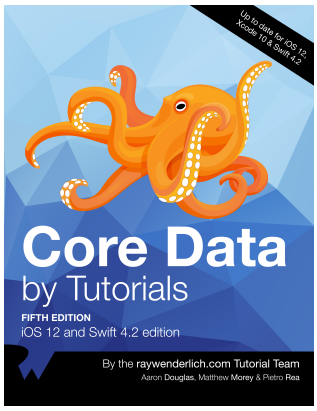
In Advanced Apple Debugging and Reverse Engineering, you'll come to realize debugging is an enjoyable process to help you better understand software. Not only will you learn to find bugs faster, but you'll also learn how other developers have solved problems similar to yours. You'll also learn how to create custom, powerful debugging scripts that will help you quickly find the secrets behind any bit of code that piques your interest. https://store.raywenderlich.com/products/advanced-apple-debugging-and-reverse-engineering

RxSwift: Reactive Programming with Swift



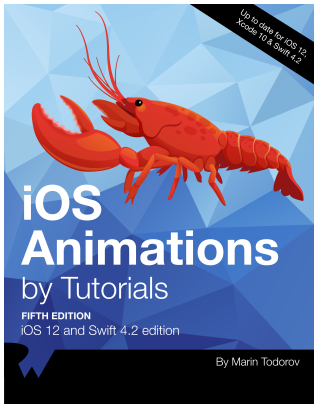
This book is for iOS developers who already feel comfortable with iOS and Swift, and want to dive deep into development with RxSwift. Start with an introduction to the reactive programming paradigm; learn about observers and observables, filtering and transforming operators, and how to work with the UI, and finish off by building a fully-featured app in RxSwift. https://store.raywenderlich.com/products/rxswift

Core Data by Tutorials



This book is for intermediate iOS developers who already know the basics of iOS and Swift 4 development but want to learn how to use Core Data to save data in their apps. Start with with the basics like setting up your own Core Data Stack all the way to advanced topics like migration, performance, multithreading, and more! https://store.raywenderlich.com/products/core-data-by-tutorials

iOS Aninmations by Tutorials



This book is for iOS developers who already know the basics of iOS and Swift 4, and want to dive deep into animations. Start with basic view animations and move all the way to layer animations, animating constraints, view controller transitions, and more! https://store.raywenderlich.com/products/ios-animations-by-tutorials

ARKit by Tutorials



Learn how to use Apple's augmented reality framework, ARKit, to build five great-looking AR apps: Tabletop Poker Dice; Immersive Sci-Fi Portal; 3D Face Masking; Location-Based Content and Monster Truck Sim. https://store.raywenderlich.com/products/arkit-by-tutorials

watchOS by Tutorials



This book is for intermediate iOS developers who already know the basics of iOS and Swift development but want to learn how to make Apple Watch apps for watchOS 4. https://store.raywenderlich.com/products/watchos-by-tutorials

tvOS by Tutorials



This book is for complete beginners to tvOS development. No prior iOS or web development knowledge is necessary, however the book does assume at least a rudimentary knowledge of Swift. This book teaches you how to make tvOS apps in two different ways: via the traditional method using UIKit, and via the new Client-Server method using TVML. https://store.raywenderlich.com/products/tvos-apprentice

Metal by Tutorials



This book will introduce you to graphics programming in Metal — Apple's framework for programming on the GPU. You'll build your own game engine in Metal where you can create 3D scenes and build your own 3D games. https://store.raywenderlich.com/products/metal-by-tutorials

Want to make games?

Learn how to make great-looking games that are deeply engaging and fun to play!

2D Apple Games by Tutorials



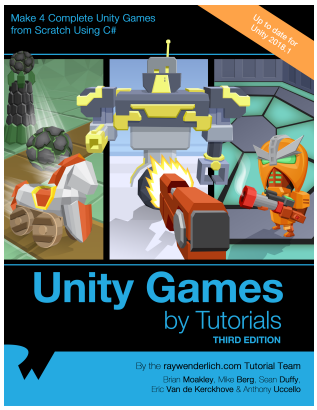
In this book, you will make 6 complete and polished mini-games, from an action game to a puzzle game to a classic platformer! This book is for beginner to advanced iOS developers. Whether you are a complete beginner to making iOS games, or an advanced iOS developer looking to learn about SpriteKit, you will learn a lot from this book! https://store.raywenderlich.com/products/2d-apple-games-by-tutorials

3D Apple Games by Tutorials



Through a series of mini-games and challenges, you will go from beginner to advanced and learn everything you need to make your own 3D game! This book is for beginner to advanced iOS developers. Whether you are a complete beginner to making iOS games, or an advanced iOS developer looking to learn about SceneKit, you will learn a lot from this book! https://store.raywenderlich.com/products/3d-apple-games-by-tutorials

Unity Games by Tutorials



Through a series of mini-games and challenges, you will go from beginner to advanced and learn everything you need to make your own 3D game! This book is for beginner to advanced iOS developers. Whether you are a complete beginner to making iOS games, or an advanced iOS developer looking to learn about SceneKit, you will learn a lot from this book! https://store.raywenderlich.com/products/unity-games-by-tutorials

Beat 'em Up Games Starter Kit



The classic beat 'em up starter kit is back — for Unity! Create your own side-scrolling beat 'em up game in the style of such arcade classics as Double Dragon, Teenage Mutant Ninja Turtles, Golden Axe and Streets of Rage. This starter kit equips you with all tools, art and instructions you'll need to create your own addictive mobile game for Android and iOS. https://store.raywenderlich.com/products/beat-em-up-game-starter-kit-unity

Want to learn Android or Kotlin?

Get a head start on learning to develop great Android apps in Kotlin, the newest first-class language for building Android apps.

Android Apprentice



The Android Apprentice takes you all the way from building your first app, to submitting your app for sale. By the end of this book, you'll be experienced enough to turn your vague ideas into real apps that you can release on the Google Play Store. The four apps you will complete will teach you how to work with the most common controls and APIs used by Android developers around the world. https://store.raywenderlich.com/products/android-apprentice

Kotlin Apprentice



This is a book for complete beginners to the new, modern Kotlin language. Everything in the book takes place in a clean, modern development environment, which means you can focus on the core features of programming in the Kotlin language, without getting bogged down in the many details of building apps. This is a sister book to the Android Apprentice.https://store.raywenderlich.com/products/kotlin-apprentice

Explore Apple code through LLDB, Python & DTrace!

Learn the powerful secrets of Apple's software debugger, LLDB, that can get more information out of any program than you ever thought possible.

In Advanced Apple Debugging and Reverse Engineering, you'll come to realize debugging is an enjoyable process to help you better understand software. Not only will you learn to find bugs faster, but you'll also learn how other developers have solved problems similar to yours. You'll also learn how to create custom, powerful debugging scripts that will help you quickly find the secrets behind any bit of code that piques your interest.

Who This Book Is For

This book is for intermediate to advanced iOS/macOS developers who are already familiar with either Swift or Objective-C and want to take their debugging skills to the next level.

Topics Covered in Advanced Apple Debugging & Reverse Engineering

- **LLDB Max Achievement:** Master LLDB and learn about its extensive list of subcommands and options.
- **1's and 0's:** Learn the low-level components available to help extract useful information from a program, from assembly calling conventions to exploring the process of dynamically-loaded frameworks.
- **The Power of Python:** Use LLDB's Python module to create powerful custom debugging commands to introspect and augment existing programs.
- **Nothing is Secret:** Learn how to use DTrace, a dynamic tracing framework, and how to write D scripts to query anything you were ever curious about on your macOS machine.
- **Case Studies:** Quickly find and solve the real-world issues that iOS and macOS developers typically face in their day-to-day development workflow.

After reading this book, you'll have the tools and knowledge to answer even the most obscure question about your code — or someone else's.

About Derek Selander

Derek is a member of the **raywenderlich.com tutorial team.** His interest with debugging grew when he started exploring how to make (the now somewhat obsolete) Xcode plugins and iOS tweaks on his jailbroken phone, both of which required exploring and augmenting programs with no source available. In his free time, he enjoys pickup soccer, guitar, and playing with his two doggies, Jake & Squid.