Mon, 07 December 2015 13:59

OS X and iOS Reverse Engineering: How to Reverse Engineer an iOS App

The aim of this article is to describe reverse engineering for OS X software and iOS apps in general terms. My goal was to provide a wide coverage of Objective-C and Swift code reversing, without going too much into details, in order to describe how to reverse engineer software.

Written by:
Konstantin Polozov,
Software Developer of Device Team

Why do we need reverse engineering?

The answer is rather simple. If we have an executable and do not have any sources, but still need to know how it works, than we need to reverse engineer it. There can be several business situations where you can apply reverse engineering legally:

- Research complicated software issues,

- Improve software compatibility with third-party solutions and formats,
- Improve the interaction between software and the platform,
- Provide easy legacy code maintenance.

You can learn more about business tasks and reverse engineering in the corresponding section.

In this article, we'll consider the following high level issues:

- How to reverse engineer OS X software and iOS apps;
- Main principles of software reverse engineering process;
- Specifics of reversing for different code types as well as different reversing objects types;
- Software reverse engineering tips.

The article consists of the following sections:

1. **"How to reverse engineer: Before you start reversing"** contains information about characteristics of executable files for OS X and iOS.
2. **"Reverse engineering tools"** considers few and far between tools for software reverse engineering.
3. **"Specifics of programming languages"** describes specifics of Objective-C and Swift code reversing.
4. **"Software reverse engineering examples and tips"** considers iOS reverse engineering examples, OS X reverse engineering examples, reversing approaches, details about OS internals.

How to reverse engineer: Before you start reversing

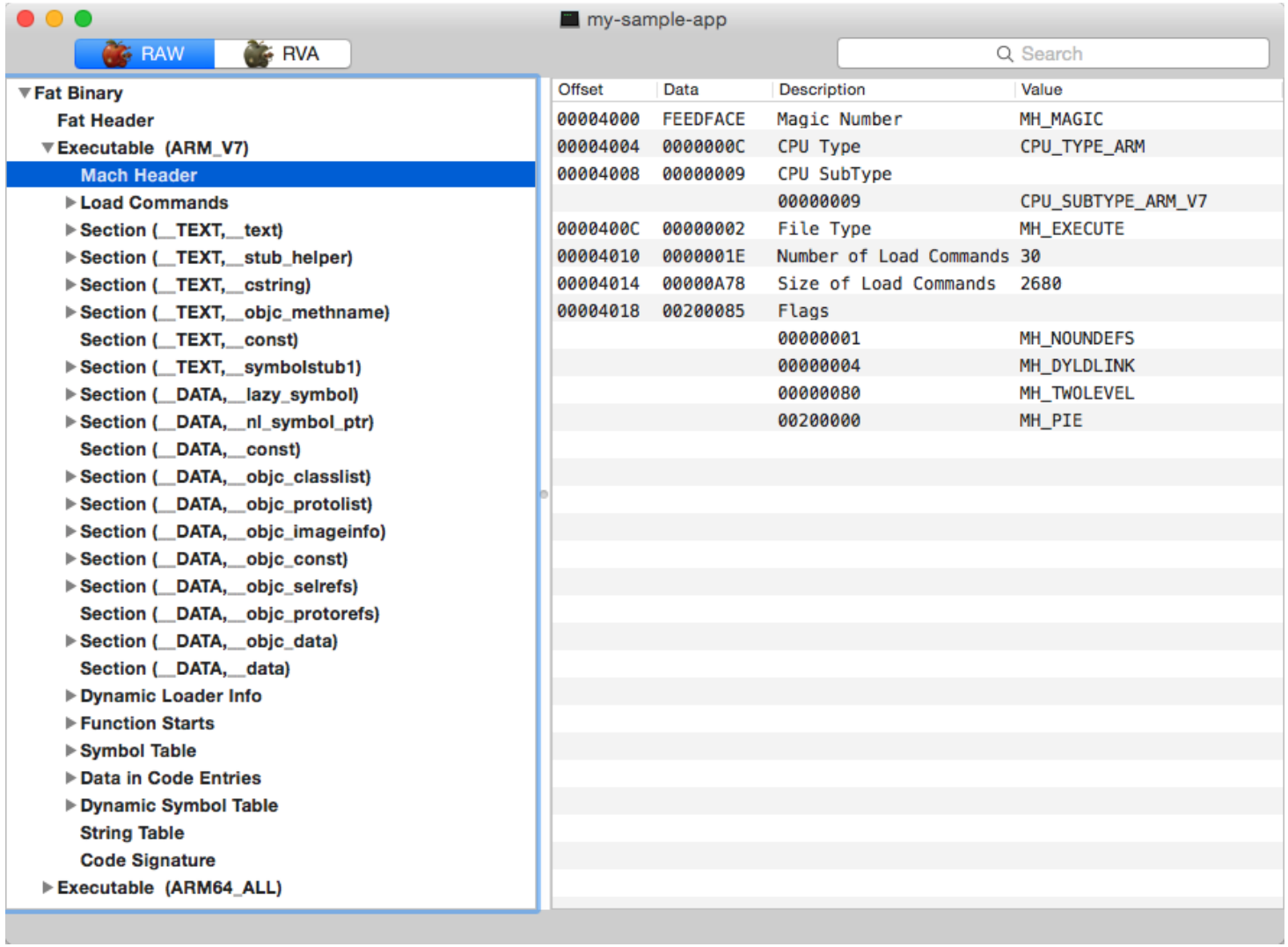
If we have finally decided to reverse engineer the binary, then we should understand that its part probably contains executable code. First of all, we need to know at least something about executable binary structure in order to learn how to reverse engineer software.

Executable binary format

In the world of Mach kernel based operating systems, it is a common thing to use Mach-O format of executables. They can be inside 'thin' or 'fat' binary files. Thin binary contains a single Mach-O executable; fat binary may contain many ones. We use fat binary to merge executable code in one single file for different CPU instruction sets.

Header is the key part of every executable for OS X or iOS. It is the first part of executable, read by loader during image loading.

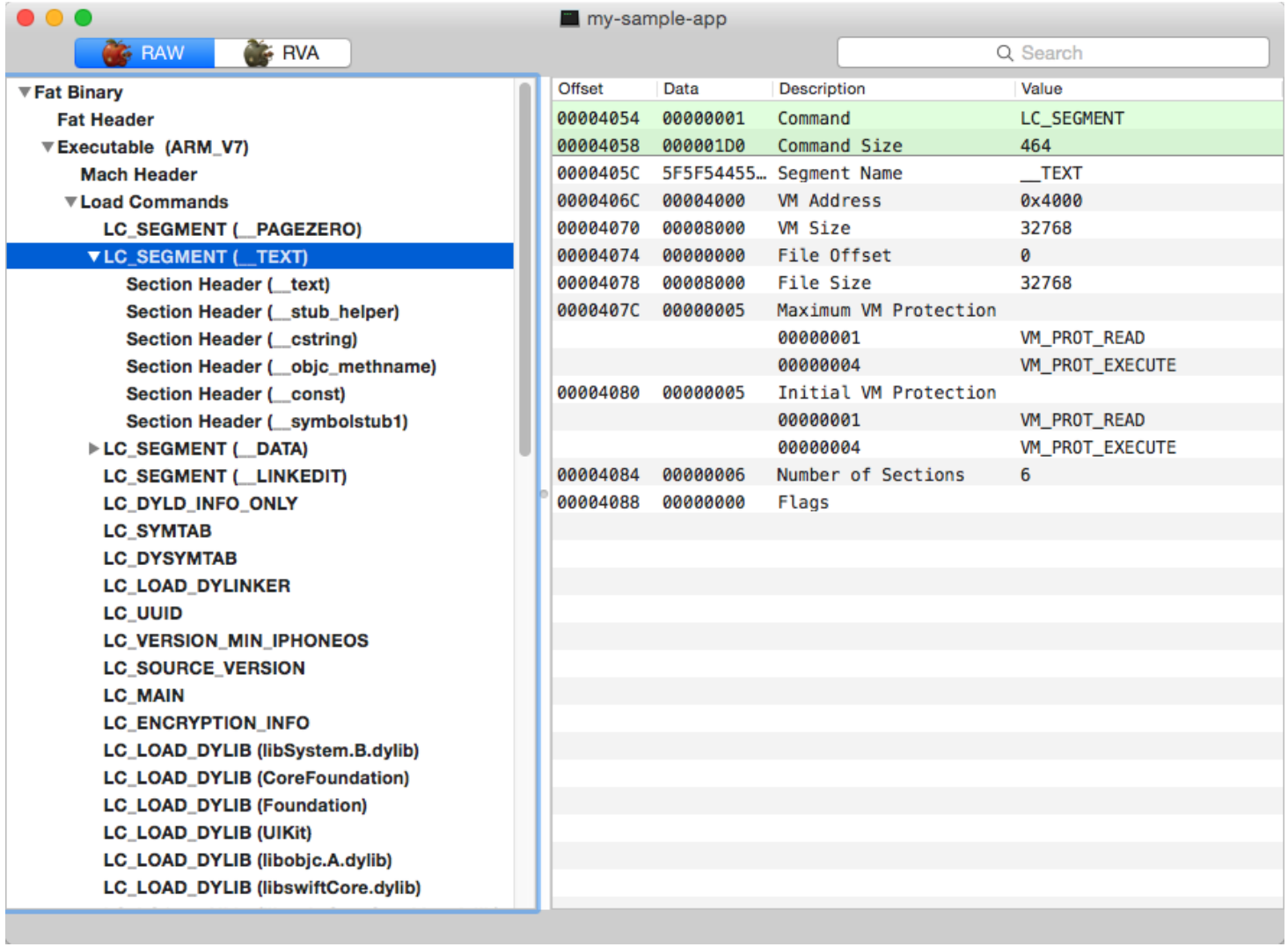
Every binary begins with a *header*. Fat binary begins with *fat header*, thin binary begins with the *mach one*. Every header starts with *magic* number, used for header identification. *Fat header* describes locations of *mach header*s for executables in binary. *Mach header* describes general info about current executable file.



It is important that mach header contains load commands. Load commands represent several things crucial for image loading:

- Segments and sections of executable and its mapping to virtual memory;
- Paths to linked dynamic libraries;
- Location of symbols tables;
- Code signature.

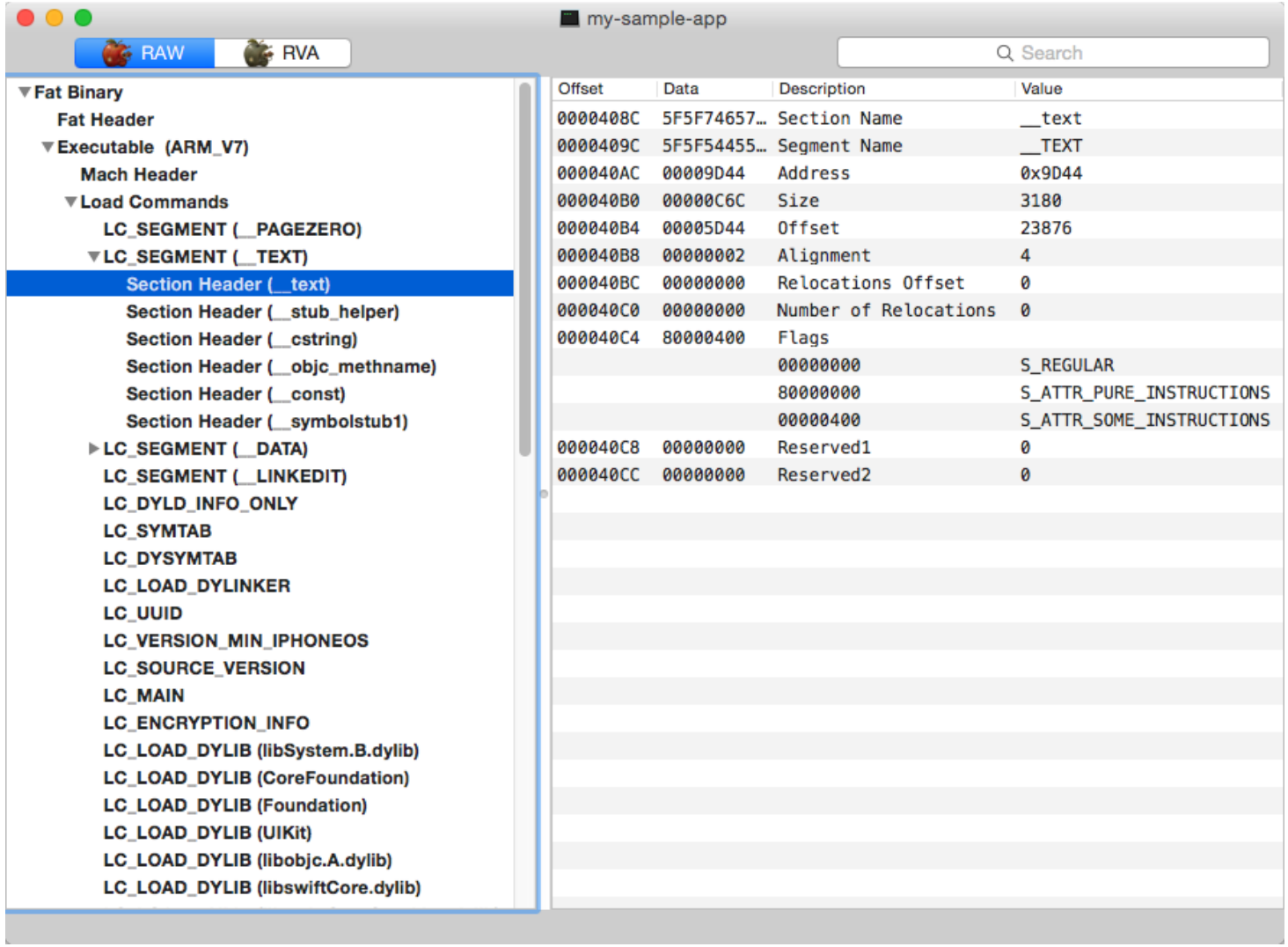
Segments are typically large pieces of executable file mapped by loader to some location in virtual address space.



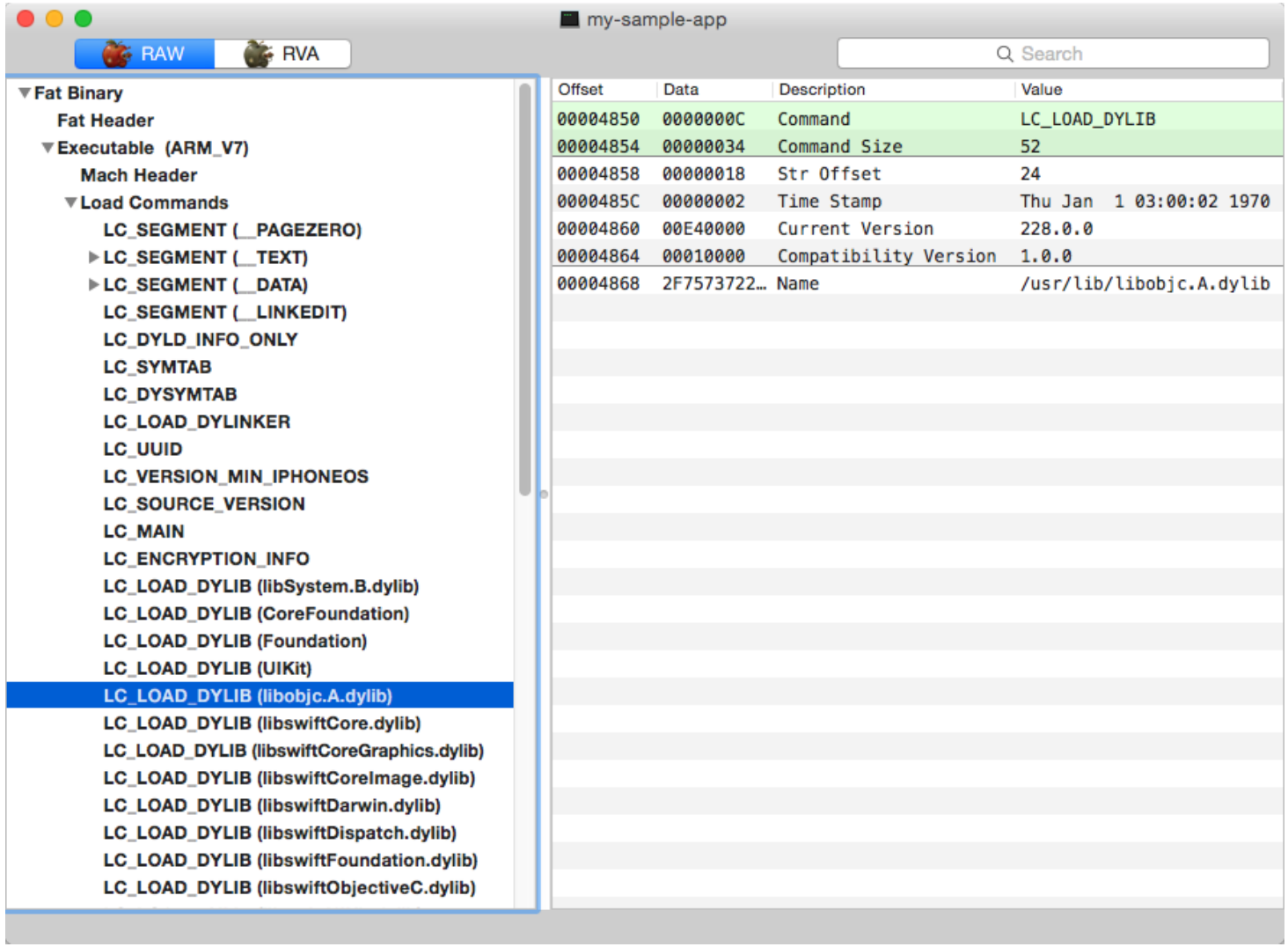
You can see a bunch of information: offset of the segment in current executable, its size, address, size of region appointed for segment mapping, and attributes.

All *segments* consist of *sections*. Section is a part of segment, intended to store some specific type of content. For example, the "__text" section of the "__TEXT" segment contains executable code, and the

"__la_symbol_ptr" section of the "DATA" segment contains table of pointers to part of external symbols called *lazy*.



Every *dynamic* library dependency is described by load command containing path to the dynamic library binary file and its version.

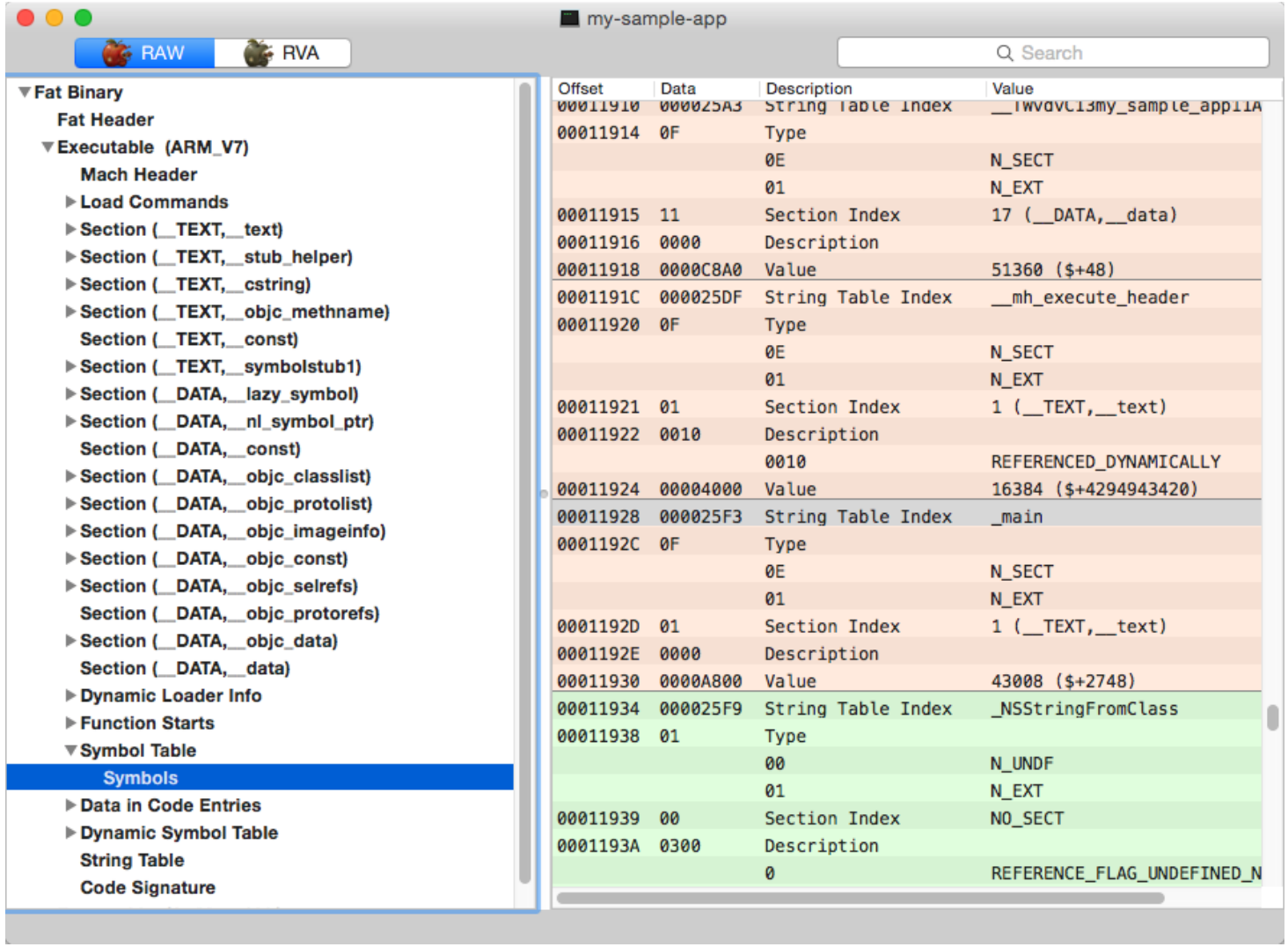


In addition, load commands contain below information, important for executable code operability:

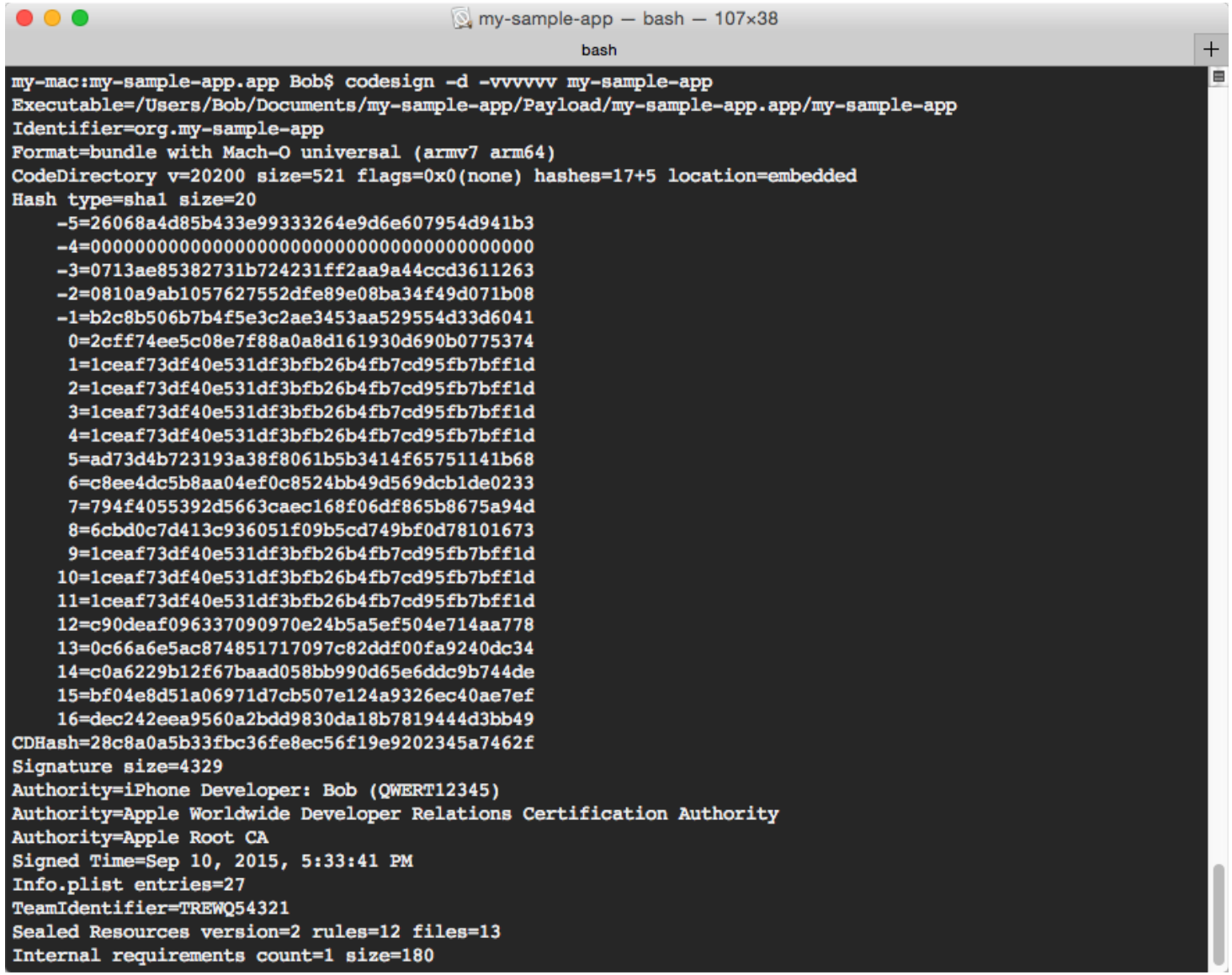
- Location of symbol tables;
- Location of import and stub tables;
- Location of table with information for dynamic loader.

There is main symbol table that contains all symbols used in current executable. Every locally or externally defined symbol, or even stub (which can be generated for external call executing through import table) is mentioned here. This table is divided into parts specifically whether the symbol is debug,

local or external. Every entry of this table represents particular part of executable code by specifying offset of its name in the string table, type, section ordinal, and other different type-specific info.



There is a string table that contains names of symbols defined in main symbol table. There is also dynamic symbols table, which links import table entries with appropriate symbol. In addition, there is one more table, which contains information used by dynamic loader for every external symbol. Code signature is a one of bad documented (but still available as open-source) parts of executable. Its contents can be displayed by means of *codesign* tool.



Code signature contains a few important things:

- Code directory;
- Code signing requirements;
- Description of sealed resource;
- Entitlements;
- Code signature.

Code directory is a structure that contains miscellaneous info (hash algorithm, size of table, size of code pages, etc.) and a table of hashes. Table itself consists of two parts: positive and negative. Positive part contains hashes of executable code pages. Negative part optionally contains hashes of code signature parts described above (requirements, resources, and entitlements) and hash of *Info.plist*.

Code *signing requirements*, *resources*, and *entitlements* are just bytestreams of appropriate files located inside bundle.

The code *signature* is an encrypted *code directory* represented in CMS format.

Architectures

Modern desktop devices usually use x86-64 CPUs. Mobile devices use armv7, armv7s and arm64 CPUs. Knowledge of instruction sets is important during reverse-engineering algorithms. In addition, it is good to be familiar with calling conventions and some things specific for arm (thumb mode, opcodes format).

Caches

Nowadays all system frameworks and dylibs are merged into a single file called *shared cache* which located at */System/Library/Caches/com.apple.dyld/*.

Software Reverse Engineering Tools

Below there are standard command-line tools for iOS app reverse engineering as well as OS X reverse engineering, available out-of-the-box on Mac:

- *lldb* is a debugger, and it is quite powerful;
- *otool* is a console tool for browsing in mach-o executables;
- *nm* is intended to browse names and symbols in mach-o executables;
- *codesign* can provide comprehensive info about Code Signature.

In addition, there are several third-party reverse engineering utilities:

- *IDA* (Interactive DisAssembler) is a must-have tool for every reverse engineer for complex research of any executable;
- *MachOView* is like *otool* and *nm*, however has GUI, and thus allows browsing structure of mach-o file in user-friendly way. It is a freeware, alas, quite unstable one;
- *class-dump* is a tool for dumping classes declarations from executable headers into normal ones;
- *Hopper* is an interactive reversing tool. Being shareware, it is available as a limited demo version.

Get details on more software reverse engineering tools in another blog post.

Specifics of programming languages

Reverse engineer Objective-C code

Objective-C is commonly used for developing applications for OS X and iOS. It relies on a specific C runtime, which is handy for reverse engineering.

Let's consider a simple code from real life:

```
NSNumber *number = [[NSNumber alloc] initWithInt:1];
```

If we compile and then decompile this code into pure C, we will get something like:

```
int v1; int v2;
v1 = objc_msgSend(_OBJC_CLASS_$_NSNUMBER, "alloc");
v2 = objc_msgSend(v1, "initWithInt", 1);
```

This example demonstrates the basics of object allocation and messaging. Every call to every method is performed by calling runtime

```
id objc_msgSend(id self, SEL op, …);
```

The first argument named *self* can be found as a pointer to an object (which obviously should be derived from *NSObject*); the second one is a pointer to such-called *selector*. These two arguments are mandatory, others are optional and contain arguments needed for method.

The runtime operates two types: SEL and IMP. They are *selector* and *implementation* respectively. The *selector* represents human-readable name of method. In the context of our example, *selector* is 'initWithInt:'. The *implementation* is a pointer to C-function and it looks like:

```
id -[NSNumber initWithInt:](id self, SEL _cmd, int value) { }
```

As we can see, this is an almost regular C-function excepting its unusual name ('+' or '-' which differs static method from *non-static*'s, the name of class, selector string) and two extra arguments (self, _cmd).

From this point of view, the main purpose of objc_msgSend is to find *implementation* for given s*elector* and object, call it passing all specified arguments.

Such approach brings at least few specific things into reverse engineering process:

- It is impossible to find direct call to a method implementation. The selector is a key for searching implementations by name.
- Human-readable selector is a great hint for understanding executable code. All selector names are resided in __objc_methname section of __TEXT segment.

Reverse engineer Swift code

Swift uses the same runtime as Objective-C, so reverse engineering of Swift code is similar to Objective-C code reversing.

iOS apps and OS X software reverse engineering examples and tips

Case 1. Reversing opensource code

Before reverse engineering anything, check the http://www.opensource.apple.com/. A lot of things are available as source code. For example, structure of Code Signature part of mach-o can be understood by inspecting codesign tool, which sources are opened for public view.

Case 2. Getting an executable to reverse engineer

The simplest case is reverse engineering of an executable from some ipa or an app. The executable can be obtained in an obvious way from app. Regarding the ipa, it is a regular zip-archive. It has a certain structure. Executable can be found inside *Payload/*.app* subdirectory of the archive. In such form, any executable can be traced by any reverse engineering tool described above.

The more tricky case is reverse engineering of the iOS part. Usually we need to have the *jailbroken* device. If *jailbreak* is not an option, there is still a possibility to get file from filesystem by using *Document Interaction* functionality.

Case 3. Reversing emulator binaries

If there is no chance to get binary from device, there is still a possibility to get it from the iOS simulator. The simulator is x86 and its code differs from iOS on real device. Nevertheless, interfaces of daemons and frameworks are the same as on true iOS.

Case 4. Finding cause of application specific issue

We usually have (or can get) a crash report and a stack trace. In such cases, we need to understand the common logic around the issue. It can look like complicated task because of private functions displayed in stack trace as ___lldb_unnamed_function. The universal way to locate such private function by using disassembler or MachOView is to find its offset relatively the __text section. We can usually get function address from trace, segment address can be found using debugger command: (lldb) image dump sections.

The great hint for understanding the internal structure of executable are tracking strings. Function names can be often tracked by strings passed to system log. The principal application delegate can be found by inspecting arguments of *NS-* or *UIApplicationMain*.

Case 5. Reverse engineering by using private or internal functionality

Usually we have a public API as a starting point and thus we know the framework we should explore. In some cases, we can use debug-symbols instead of framework binary. We can find them at *~/Library/Developer/Xcode/iOS DeviceSupport/*.

It is expected, that internals are not exported. Regarding Objective-C code, even internal code can be executed by using the low-level obj-c runtime. If it is not, than it is always a possibility to dump classes declarations using otool or class-dump and use them without confusing the linker.

Case 6. Communicating with daemon

It is a common thing, when framework appears to be a proxy between the application and a daemon. The example of such client-server tandem is *MobileInstallation.framework* and *installd*. When someone makes a call to *MobileInstallation.framework*, it delegates most of the work to *installd* using *rpc*.

The first *rpc* OS X and iOS use is *mach* interprocess communication facility. The second one is *xpc* that also uses *mach* messages behind the scenes, but it is much more high-level.

The *xpc* runs in restricted environment by default. Any capabilities must be whitelisted by a set of entitlements. In other case privileged tasks are not permitted. This often makes *xpc* hard to use. That's

why low-level *mach* is the better option.

Resources

- http://www.iphonedevwiki.net
- https://www.theiphonewiki.com
- http://nshipster.com
- http://www.opensource.apple.com

Reverse engineering tools

- http://lldb.llvm.org
- https://www.hex-rays.com/products/ida
- http://sourceforge.net/projects/machoview
- https://github.com/gdbinit/MachOView
- http://www.hopperapp.com
- http://stevenygard.com/projects/class-dump

Other useful materials

- https://developer.apple.com/library/mac/documentation/DeveloperTools/Conceptual/MachORuntime
- https://developer.apple.com/library/mac/documentation/Cocoa/Reference/ObjCRuntimeRef

Literature

- Mac OS X and iOS Internals: To the Apple's Core, Jonathan Levin
- Mac OS X Internals: A Systems Approach, Amit Singh
- iOS Hacker's Handbook, Charlie Miller

Ready to hire experienced reverse engineering team to work on your OS X and iOS projects? Just contact us and we will provide you all details!