DYLD Detailed

Jonathan Levin, http://newosxbook.com/ - 8/12/13

1. About

While maintaining and adding more functionality to JTool, I found myself deeply bogged down in implementing support for Mach-O's LINKEDIT sections, LC_SYMTAB, and other arcane and relatively undocumented corners of DYLD. Add to that, DYLD has been relatively skimmed in my book [*], and not much in that of my predecessor. Scouring the Internet with Google finds only one decent reference[1], though it's woefully incomplete and basically just rehashes stuff from the book. Needless to say Apple makes no effort to provide documentation outside its "Mach-O Programming Topics"[2] document, which is by now very dated. What better way, then, to right a wrong and shed some light on it, than an article?

Why should you care? (Target Audience)

I said so in the book, and I'll state it again - There is no knowledge that is not power, and in the case of linking - we're talking about a lot of power. Virtually every binary run in OS X or iOS is dynamically linked, and being able to intervene in the linking process bestows significant capabilities - function interception, auditing and hooking, being the most important ones. Reverse engineers, security-oriented developers (i.e. Anti-Malware) and hackers will hopefully find this information very useful. It should be noted that dyld allows for hooking and interception via environment variables - most notably DYLD_INSERT_LIBRARIES (akin to ld's LD_PRELOAD) and DYLD_LIBRARY_PATH (like ld's LD_LIBRARY_PATH), and its function interposing mechanism. These are covered in the book (somewhere in Chapter 4, with a demo on this website[3]), and are therefore not discussed in this document.

Prerequisite: About Linking

Nearly all binaries, in UN*X and Windows systems alike, are dynamically linked. The benefits of dynamic linking are many, and include:

- Code reuse: commonly used code can be extracted to a library, which is then shared by many clients
- Easy updating: code residing in a library can easily be updated, and the library replaced, so long as the symbols are by and large the same. A classic example of this can be seen in Windows' "CreateWindow", which creates totally different-looking windows for the same application throughout Windows versions (think Win95 vs. XP vs. 7-8). The developer merely says "CreateWindow", not knowing how the window gets created. The OS does the rest, and different versions of the OS may do so differently.
- Reducing disk usage: as commonly used code now has only one copy, as every single binary which uses it.
- Reducing RAM usage: is by far, the most important advantage: A single all processes, thereby only getting hit by the library's RAM usage once. T only, executable), so the same physical copy is implicitly shared by many immense amounts of memory, especially in RAM-challenged systems like Android.

UN*X, whose de-facto standard format is ELF, uses ld(1) as the program linker-loader, and the ".so" (shared object) files for libraries. OS X, thinking differently, uses ".dylib" (dynamic library) files. The standard nm(1) command is still supported, as are the dl* APIs (dlopen(3), dlsym(3), etc) - but the implementations are radically different (as is the nomenclature - what ld(1) calls "sections", DYLD calls "segments", and further divides into sections). DYLD's source code is open, but makes for a terrible read. DYLD offers many of the classic ld(1) functions, and then some.

Nomenclature

Throughout this article, the following terms are used:

- dylib: A dynamic library. Akin to a UN*X shared object. A Mach-O object of type MH_DYLIB (0x6), loaded into other executables by the LC_LOAD_DYLIB (0xc) Mach-O command or the dlopen(3) API. For the record, it's worth noting that OS X also supports the concept of a fixed library (A Mach-o object of type MH_FVMLIB (0x3) loaded into other executables by the LC_LOADFVMLIB (0x6) command. Fixed libraries, however, are virtually extinct.
- symbol: A variable or function in a Mach-O file which may or may not be visible outside that file.
- binding: Connecting a symbol reference to its address in memory. Binding may be load-time, lazy (deferred) or (missing/overridable). These can be controlled at compile time: ld's -bind_at_load specifies load-time binding, and __attribute((weak_import)) for weak symbols. There is also an option to prebind libraries to fixed addresses (-prebind switch of ld)

Tools:

Apple provides otool(1), dyldinfo(1) and pagestuff(1) - if you have Xcode. If you don't, or - if you want to analyze Mach-O binaries on Linux - you are welcome to use JTool instead (http://www.newosxbook.com/files/jtool.tar). This is an all-in-one replacement for the above tools, with far more capable features, including an experimental disassembler. The tar file contains an OS X and iOS version bundled into one universal binary, as well as an ELF version (for Linux 64-bit). It's free to download and use, and will remain so.

In the outputs shown, I've color coded: white is what you should type. yellow is for my own annotations. Everything else is verbatim the output of the commands.

Calling external functions

If you disassemble any Mach-O dynamically linked binary, you will no doubt see, sooner or later, a call to an external function, supplied by some library (commonly, libSystem.B.dylib). These calls are implemented as calls to the Mach-O's symbol stub section. Consider the following example, from OS X's /bin/ls:

```
morpheus@Zephyr (~)$ otool -tV /bin/ls | grep stub

0000000100000ab5        jmpq    0x100003fea ## symbol stub for: _strcoll
..
0000000100000e49        callq   0x100003fd8 ## symbol stub for: _setlocale
0000000100000e59        callq   0x100003f84 ## symbol stub for: _isatty
0000000100000e73        callq   0x100003f4e ## symbol stub for: _getenv
0000000100000e85        callq   0x100003ef4 ## symbol stub for: _atoi
0000000100000e9c        callq   0x100003f7e ## symbol stub for: _ioctl
0000000100000eca        callq   0x100003f4e ## symbol stub for: _getenv
0000000100000ed7        callq   0x100003ef4 ## symbol stub for: _atoi
0000000100000ee8        callq   0x100003f6c ## symbol stub for: _getuid
0000000100000f40        callq   0x100003f5a ## symbol stub for: _getopt
0000000100001073        callq   0x100003efa ## symbol stub for: _compat_mode
00000001000010b1        callq   0x100003fd2 ## symbol stub for: _setenv
..
0000000100003e6c        callq   0x100003f42 ## symbol stub for: _fwrite
0000000100003e76        callq   0x100003f06 ## symbol stub for: _exit

morpheus@Zephyr (~)$ jtool -l -v /bin/ls | grep stubs
   Mem: 0x100003e7c-0x10000403e File: 0x00003e7c-0x0000403e             __TEXT.__stubs
```

Following on the experiment from page 116**, If you have gdb or lldb (as of Xcode 5), you can use either to examine the contents of this "stub" section:


```
morpheus@Zephyr (~) $ /Developer/usr/bin/lldb /bin/ls
Current executable set to '/bin/ls' (x86_64).
(lldb) x/i 0x100003fea
0x100003fea:  ff 25 30 12 00 00  jmpq   *4656(%rip)
(lldb) b 0x100003fea
Breakpoint 1: address = 0x0000000100003fea
(lldb) r  # run the process, to hit the breakpoint
Process 1671 launched: '/bin/ls' (x86_64)
Process 1671 stopped
* thread #1: tid = 0x18105, 0x0000000100003fea ls`strcoll, queue = 'com.apple.main-thread
    frame #0: 0x0000000100003fea ls`strcoll
ls`symbol stub for: strcoll:
-> 0x100003fea:  jmpq   *4656(%rip)               ; (void *)0x00000001000042b2

# Ok.. let's see what lies in 42b2...
(lldb) x/2i 0x00000001000042b2
0x1000042b2:  68 60 04 00 00  pushq  $1120
0x1000042b7:  e9 84 fd ff ff  jmpq   0x100004040
```

The book goes on (till page 121) to explain how DYLD manages the stubs, and populates them with the actual addresses of the functions, using dyld_stub_binder. It does not, however, explain HOW that's done. This is what we'll discuss here. But before we do, a bit about LINKEDIT:

DYLD_INFO and LINKEDIT

Starting with OS X 10.5 or 10.6, Apple decided to implement a special segment in Mach-O files for DYLD's usage. This segment, traditionally called __LINKEDIT, consists of information used by DYLD in the process of linking and binding symbols. This section is (for the most part) meaningful only to DYLD - the kernel is completely oblivious to its presence.

DYLD relies on a special load command, DYLD_INFO, to serve as a "table of contents" for the segment. This can be seen with otool(1) or jtool:

```
$  jtool -l -v /bin/ls
...

LC 03: LC_SEGMENT_64          Mem: 0x100006000-0x100009000      File: 0x6000-0x87a0
LC 04: LC_DYLD_INFO_ONLY
           Rebase info: 24    bytes at offset 24576 (0x6000-0x6018)
           Bind info:   104   bytes at offset 24600 (0x6018-0x6080)
           Lazy info:   1352  bytes at offset 24704 (0x6080-0x65c8)
        No Weak info
           Export info: 32    bytes at offset 26056 (0x65c8-0x65e8)
LC 05: LC_SYMTAB                Symbol table is at offset 0x66c4, with 83 entries
...
```

Using jtool -v -l on a binary to display load commands, with a focus on the __LINKEDIT segment

Jtool contains a useful option, --pages, which presents a mapping of the Mach-O regions (segments, sections, and load command data), somewhat similar to (but more detailed than) pagestuff(1). This can be used, among other things, to dump the contents of __LINKEDIT:

```
$  jtool --pages /bin/ls
bash-3.2# jtool --pages /bin/ls
0x0-0x0 __PAGEZERO
0x0-0x5000       __TEXT
0x5000-0x6000    __DATA
0x6000-0x87a0    __LINKEDIT
        0x6000-0x6018   Rebase Info
        0x6018-0x6080   Binding Info
        0x6080-0x65c8   Lazy Bind Info
        0x65c8-0x65e8   Exports
        0x65e8-0x6620   Function Starts
        0x6620-0x66c4   Code Signature DRS
        0x66c4-0x6bf4   Symbol Table
        0x6bf4-0x6e68   Indirect Symbol Table
        0x6e68-0x7230   String Table
        0x7230-0x87a0   Code signature
```

Using jtool --pages on a sample binary

As can be seen from the above output, the general layout of the __LINKEDIT i

Indexed by LC_DYLD_INFO	Rebase Info	Image rebase info - contains rebasing opcodes
	Bind Info	Image symbol binding info for required import symbols
	Lazy Bind Info	Image symbol binding info for lazy import symbols. This will be 0 for binaries compiled with ld's -bind_at_load
	Weak Bind Info	Image symbol binding info for weak import symbols
	Export Info	Image symbol binding info for symbols exported by this image
Pointed to by LC_SEGMENT_SPLIT_INFO	Segment Split, if any	Segment split information
Pointed to by LC_FUNCTION_STARTS	Function start information	Function start point information (ULEB128)
Pointed to by LC_DATA_IN_CODE	Data regions in code	Data region information (ULEB128)
Pointed to by	Code	Code signing DRs of dependent dylibs

LC_CODE_SIGN_DRS	Signing DRs	
Pointed to by LC_SYMTAB	Symbol Table	Table of symbols, in nlist format
Pointed to by LC_DYSYMTAB	Indirect Symbol Table	Table of indirect symbols
	String Table	Array of symbol names
Pointed to by LC_CODE_SIGNATURE	Code Signature	Code Signing blob (discussed in a future article)

Layout of __LINKEDIT segment

DYLD makes extensive use of the ULEB128 encoding, which is (in the author's humble opinion) a crude and stingy encoding method. Low level implementors would be wide to familiarize themselves with the encoding, which is also used in DWARF and other binary-related formats.

DYLD OpCodes

DYLD uses a special encoding - consisting of various "opcodes" - to store and load symbol binding information. These opcodes are used to populate the rebase information and binding tables pointed to by the LC_DYLD_INFO command. There are two types of opcodes: Rebasing opcodes and Binding opcodes.

Binding opcodes

Binding opcodes (used for both lazy and non-lazy symbols) are defined in as BIND_xxx constants:

DONE	0x00	End of opcode list
SET_DYLIB_ORDINAL_IMM	0x10	Set dylib ordinal to immediate (lower 4-bits). Used for ordinal numbers from 0-15
SET_DYLIB_ORDINAL_ULEB	0x20	Set dylib ordinal to following ULEB128 encoding. Used for ordinal n[illegible]
SET_DYLIB_SPECIAL_IMM	0x30	Set dylib [illegible] the value [illegible] Currently [illegible] • BIND_SPECIAL_DYLIB_SELF (0) • BIND_SPECIAL_DYLIB_MAIN_EXECUTABLE(-1) • BIND_SEPCIAL_DYLIB_FLAT_LOOKUP(-2)
SET_SYMBOL_TRAILING_FLAGS_IMM	0x40	Set the following symbol (NULL-terminated char[]). The flags (in the immediate value) can be either BIND_SYMBOL_FLAGS_WEAK_IMPORT(0) or BIND_SYMBOL_FLAGS_NON_WEAK_DEFINITION(8).
SET_TYPE_IMM	0x50	Set the type to immediate (lower 4-bits). Known types are: • TYPE_POINTER (most common) • TYPE_TEXT_ABSOLUTE32 • TYPE_TEXT_PCREL32
SET_ADDEND_SLEG	0x60	Set the addend field to the following SLEB128 encoding.
SET_SEGMENT_AND_OFFSET_ULEB	0x70	Set Segment to immediate value, and address to the following SLEB128 encoding
ADD_ADDR_ULEB	0x80	Set the address field to the following SLEB128 encoding.
DO_BIND	0x90	Perform binding of current table row
DO_BIND_ADD_ADDR_ULEB	0xA0	Perform binding, also add following ULEB128 as address
DO_BIND_ADD_ADDR_IMM_SCALED	0xB0	Perform binding, also add immediate (lower 4-bits) using scaling
DO_BIND_ADD_ADDR_ULEB_TIMES_SKIPPING_ULEB	0xC0	Perform binding for several symbols (as following ULEB128), and skip several bytes (as the ULEB128 which follows next). Rare.

Each opcode is specified in the topmost 4-bits (e.g. BIND_OPCODE_MASK (0xF0) in . Arugments to opcodes are either the "immediate" values in the lower 4-bits (for those with _IMM), or follow the opcode byte in ULEB128 notation for integers, or a character array (SET_SYMBOL_TRAILING_FLAGS_IMM).

The opcodes populate the individual columns of row entries in the binding tables, with each row terminated by a DO_BIND. Each row carries by default the values of the previous row, and so an opcode is specified only if the column value is changed in between two symbols. This allows for table compression. The tables are a little bit different between the binding symbols (bind info) and the lazy binding symbols (lazy_bind info):

```
bind information:
segment section          address        type    addend dylib            symbol

lazy binding information (from lazy_bind part of dyld info):
segment section          address    index  dylib
```

For example, consider the following output from jtool (or dyldinfo) -opcodes, annotated:

```
0x0000 BIND_OPCODE_SET_DYLIB_ORDINAL_IMM(3)                                   # Sets DYLI
0x0001 BIND_OPCODE_SET_SYMBOL_TRAILING_FLAGS_IMM(0x00, __DefaultRuneLocale)   # Sets symb
0x0016 BIND_OPCODE_SET_TYPE_IMM(1)                                            # Sets type
0x0017 BIND_OPCODE_SET_SEGMENT_AND_OFFSET_ULEB(0x02, 0x00000000)              # Sets segm
0x0019 BIND_OPCODE_DO_BIND()                                                  # Row done

#
# The second row will inherit all the values from the first, but override symbol name:
#

0x001A BIND_OPCODE_SET_SYMBOL_TRAILING_FLAGS_IMM(0x00, ___stack_chk_guard)    # Sets symb
0x002E BIND_OPCODE_DO_BIND()

#
# Again, third row inherits all the values from second, save symbol name:

0x002F BIND_OPCODE_SET_SYMBOL_TRAILING_FLAGS_IMM(0x00, ___stderrp)
0x003B BIND_OPCODE_DO_BIND()
..
```

The opcodes are used by our special friend, dyld_stub_binder, as we discuss later. But before we can get to it, we have to make another segue to explain the two types of symbol tables in Mach-O.

Symbol Tables

The Symbol Table (LC_SYMTAB)

The Symbol Table in a Mach-O file is described in an LC_SYMTAB command

```
struct symtab_command {
        uint32_t        cmd;            /* LC_SYMTAB */
        uint32_t        cmdsize;        /* sizeof(struct symtab_command) */
        uint32_t        symoff;         /* symbol table offset */
        uint32_t        nsyms;          /* number of symbol table entries */
        uint32_t        stroff;         /* string table offset */
        uint32_t        strsize;        /* string table size in bytes */
};
```

This can be seen with jtool, using -l:

```
LC 05: LC_SYMTAB                Symbol table is at offset 0x66c4, with 83 entries
                                String table is at offset 6e68, 968 bytes
```

The symbol table itself is an array of nsyms entries, each a struct nlist or struct nlist_64 - depending on the file type (MH_MAGIC or MH_MAGIC_64, respectively). The nlist structures follow the BSD format, with some minor modifications. The String Table is nothing more than an array of NULL-terminated strings, which follow one another

The Indirect Symbol Table (LC_DYSYMTAB)

The Indirect Symbol Table in a Mach-O file is described in an LC_DYSYMTAB command. This command details (among other things) the offset of this table, and the number of symbols it contains. This can be seen with otool (or jtool) -l, as follows:

```
..
LC 06: LC_DYSYMTAB
          1 local symbols at index     0
          1 external symbols at index  1
         81 undefined symbols at index 2
            No TOC
            No modtab
```

```
        157 Indirect symbols at offset 0x6bf4
..
```

The indirect symbol table is, in fact, nothing more than an array of indices into the main symbol table (the one pointed to by LC_SYMTAB). Dumping the indirect symbol table is straightforward with jtool, by specifying an offset (or address) inside the table:

```
morpheus@Zephyr (~)$ jtool -do 0x6bf8 /bin/ls
Dumping from offset 0x6bf8 (address 0x100006bf8, Segment: __LINKEDIT)
Offset of address specified (100006bf8) falls within Indirect Symbol Table - dumping from
Entry 1: 0000002c        _humanize_number
Entry 2: 00000047        _tgetent
Entry 3: 00000048        _tgetstr
Entry 4: 00000049        _tgoto
Entry 5: 0000004b        _tputs
Entry 6: 00000003        ___assert_rtn
Entry 7: 00000004        ___error
Entry 8: 00000005        ___maskrune
...
```

The indirect symbol table is used with two specific Mach-O sections - the __DATA.__nl_symbol_ptr, and __DATA.__lazy_symbol. We discuss these next.

__DATA.__nl_symbol_ptr and __DATA.__lazy_symbol

The __DATA.__nl_symbol_ptr section contains the "non-lazy" symbol pointers. Recall, that binding of symbols can be performed either at load time, or on first use. The "non lazy" pointers are those which must be bound at load time (that is, if binding is unsuccessful, the binary will fail to load). The name of the section is somewhat of a convention, but it is the section type (0x06 - S_NON_LAZY_SYMBOL_POINTERS) which defines its contents. As for the section contents, they are detailed in <mach-o/loader.h> as follows:

```
/*
 * For the two types of symbol pointers sections and the symbol stubs section
 * they have indirect symbol table entries.  For each of the entries in the
 * section the indirect symbol table entries, in corresponding order in the
 * indirect symbol table, start at the index stored in the reserv
 * of the section structure.  Since the indirect symbol table ent
 * correspond to the entries in the section the number of indirec
 * entries is inferred from the size of the section divided by th
 * entries in the section.  For symbol pointers sections the size
 * in the section is 4 bytes and for symbol stubs sections the by
 * stubs is stored in the reserved2 field of the section structur
 */
#define S_NON_LAZY_SYMBOL_POINTERS      0x6     /* section with only non-lazy
                                                   symbol pointers */
#define S_LAZY_SYMBOL_POINTERS          0x7     /* section with only lazy symbol
                                                   pointers */
#define S_SYMBOL_STUBS                  0x8     /* section with only symbol
                                                   stubs, byte size of stub in
                                                   the reserved2 field */
```


It is worth mentioning that __nl_symbol_ptr is not the only "non-lazy" section: The binary's Global Offset Table (GOT) is in its own section, __DATA.__GOT, similarly marked with S_NON_LAZY_SYMBOL_POINTERS. It's also noteworthy that only one of these values is held in the section's flags field (which erroneously implies these are bit-flags - they are not, but there are some higher bit flags which may be or'ed with these values). The __DATA.__lazy_symbol section contains lazy symbols. These are symbols which will be bound on first use. The code to do so is in an additional section, referred to as the *symbol stubs*. The "stubs" consist of boilerplate code, which is naturally architecture dependent. Apple Developer's "OS X Assembler Reference"[4] details this well, but unfortunately only for the deprecated PowerPC architecture. JTool's disassembler is almost fully functional for ARM (but still very partial for x86_64). We therefore show the ARMv7 (iOS) case next.

dyld_stub_binder and _helper (in iOS)

Stub resolution in iOS and OS X is practically the same. The __TEXT.__stub_helper contains a single function, which sets up a call to the dyld_stub_binder according to the value pointed to by R12, a.k.a the Intra-Procedural register***. The other entries in stub_helper are trampolines to this function, each setting up R12 to hold the value of the indirect symbol table entry corresponding to the function to be bound. This is shown in the annotated jtool disassembly of ScreenShotr (the screen capture utility used by Xcode, from iOS's DeveloperDiskImage.dmg), below:

```
morpheus@Zephyr(~)$ jtool -dA __TEXT.__stub_helper  ~/Documents/RE/ScreenShotr  # note -dA
Disassembling from file offset 0x18d4, Address 0x28d4
```

```
28d4     e52dc004          PUSH   IP                ; STR IP, [ SP, #-4 ]!              # PUSHes
28d8     e59fc010          LDR    IP, [PC, 16]      ; R12 = *(28f0) = 0x7f8             # Load
28dc     e08fc00c          ADD    IP, PC, IP        ; R12 = 0x30dc                      # Correct
28e0     e52dc004          PUSH   IP                ; STR IP, [ SP, #-4 ]!              # PUSHes

28e4     e59fc008          LDR    IP, [PC, 8]       ; R12 = *(28f4) = 0x7e8             # Load
28e8     e08fc00c          ADD    IP, PC, IP        ; R12 = 0x30d8                      # Correct
28ec     e59cf000          LDR    PC, [IP, 0]       ; R15 = *(30d8) dyld_stub_binder # goto dy

28f0     7f8               DCD    0x7f8             # Offset of 0x30dc, PC-relative
28f4     7e8               DCD    0x7e8             # Offset of dyld_stub_binder, PC-relative
---------------------------------------------
28f8     e59fc000          LDR    IP, [PC, 0]       ; R12 = *(2900) = 0x0  # Lazy binding opco
28fc     eafffff4          B      0xffffffd0        ; 0x28d4               # Jump to stub_hand
2900     0                 DCD    0
---------------------------------------------
2904     e59fc000          LDR    IP, [PC, 0]       ; R12 = *(290c) = 0x17 # Lazy binding opco
2908     eafffff1          B      0xffffffc4        ; 0x28d4               # Jump to stub_hand
290c     17                DCD    0x17
---------------------------------------------
...

morpheus@Zephyr(~)$ jtool -opcodes  ~/Documents/RE/ScreenShotr  # Can also use dyldinfo -o
..
lazy binding opcodes:
0x0000 BIND_OPCODE_SET_SEGMENT_AND_OFFSET_ULEB(0x02, 0x00000000)
0x0002 BIND_OPCODE_SET_DYLIB_ORDINAL_IMM(2)
0x0003 BIND_OPCODE_SET_SYMBOL_TRAILING_FLAGS_IMM(0x00, _IOSurfaceCreate)
0x0015 BIND_OPCODE_DO_BIND()
0x0016 BIND_OPCODE_DONE
0x0017 BIND_OPCODE_SET_SEGMENT_AND_OFFSET_ULEB(0x02, 0x00000004)
0x0019 BIND_OPCODE_SET_DYLIB_ORDINAL_IMM(2)
0x001A BIND_OPCODE_SET_SYMBOL_TRAILING_FLAGS_IMM(0x00, _IOSurfaceGetBaseAddress)
0x0034 BIND_OPCODE_DO_BIND()
0x0035 BIND_OPCODE_DONE
0x0036 BIND_OPCODE_SET_SEGMENT_AND_OFFSET_ULEB(0x02, 0x00000008)
0x0038 BIND_OPCODE_SET_DYLIB_ORDINAL_IMM(2)
0x0039 BIND_OPCODE_SET_SYMBOL_TRAILING_FLAGS_IMM(0x00, _IOSurfaceLock)
0x0049 BIND_OPCODE_DO_BIND()
0x004A BIND_OPCODE_DONE
0x004B BIND_OPCODE_SET_SEGMENT_AND_OFFSET_ULEB(0x02, 0x0000000C)
0x004D BIND_OPCODE_SET_DYLIB_ORDINAL_IMM(2)
0x004E BIND_OPCODE_SET_SYMBOL_TRAILING_FLAGS_IMM(0x00, _IOSurfaceUnlock)
0x0060 BIND_OPCODE_DO_BIND()
```


dyld_stub_binder is exported by libSystem.B.dylib, though in actuality it is a re
Using Jtool again, we can see:

```
morpheus@Zephyr(~)$ ARCH=armv7s jtool -dA dyld_stub_binder /Developer/Platforms/iPhoneOS.p
dyld_stub_binder:
# coming into this function, we have two arguments on the stack:
# *SP      = Offset into bind information
# *(SP+4) = 0x30c4 - Address of image loader cache
10cc     e92d408f          PUSH   {R0,R1,R2,R3,R7,LR} ; SP -= 24           # save registers
10d0     e28d7010          ADD    R7, SP, #0x10       ; R7 = SP + 0x10 (point to previous R7)
10d4     e59d0018          LDR    R0, [SP, 24]        ; R0 = *(SP + 0x18) = *(Initial_SP)
10d8     e59d101c          LDR    R1, [SP, 28]        ; R1 = *(SP + 0x1c) = *(Initial_SP + 4
10dc     fa0001ef          BLX    0x7bc               ; 0x18a0 __Z21_dyld_fast_stub_entryPvl
10e0     e1a0c000          MOV    IP, R0              ;
; IP = dyld_fast_sub_entry (void *, long)
10e4     e8bd408f          POP    {R0,R1,R2,R3,R7,LR}                      # restore registe
10e8     e28dd008          ADD    SP, SP, #0x8        ; SP  = 0x8          # Clear stack
10ec     e12fff1c          BX     IP                                       # Jump to bound sy
```

Jtool's disassembly is corroborated by DYLD's source, which surprisingly enough contains an #if __arm__ statement for iOS [5] which Apple has not removed. If you're following with x86_64 (e.g. with /bin/ls), the 0x100004040 from the lldb example is the trampoline to dyld_stub_binder. In other words, the code will look something like this when you break on 0x100004040:

```
* thread #1: tid = 0x185f7, 0x0000000100004040 ls, queue = 'com.apple.main-thread, stop re
    frame #0: 0x0000000100004040 ls
# stack already contains the offset into the LINKEDIT bind information, which is different
# When we get here, this is common code, and we further push the address of the cache:
-> 0x100004040:  leaq   4073(%rip), %r11          ; (void *)0x0000000000000000
   0x100004047:  pushq  %r11
```

```
0x100004049:  jmpq   *4057(%rip)              ; (void *)0x00007fff8c80e878: dyld_stub_
0x10000404f:  nop
```

Hopefully, this fills in the missing pieces, showing you not just what symbols are bound, but HOW they are bound. I hope to provide more information about LINKEDIT (specifically, the juicy parts of codesigning. You are always welcome to go online at the Book Forum and comment, ask questions, etc.

References:

1. MikeAsh.com, article on DYLD by G. Raskind
2. Apple Developer - Mach-O Programming Topics
3. Source code of DYLD Interpose example from the book
4. Apple Developer - OS X Assembler Reference
5. http://opensource.apple.com/source/dyld/dyld-210.2.3/src/dyld_stub_binder.s - Source of DYLD's stub_binder, for both x86_64 and ARM

Footnotes

* (something I heard several times already by now as a criticism is a "lack of detail" - considering that Wiley restricted the book originally to 500 pages, I'm very lucky to have been able to extend it to the 800 pages it is - but some things just had to be left out, folks.. which is why I'm providing lots of extra content on the website..)

** - While we're on the subject, there's a typo in page 116 (should be "using Xcode's dyldinfo(1) **or** nm(1). One of the all too many omissions and editorial mistakes inserted, ironically, by the copy editor. Incidentally, nm(1) only shows the symbols, not where they are located. You might want to try jtool's -S feature (cloning nm(1)) with -v.

*** - This is a register which the ARM ABI allows for use in between functions/procedures.