Building a Custom Mach-O Memory Loader for macOS - Part 1

2023-02-04

Posted on

In the last post we looked at how we could patch dyld to restore in-memory execution. One of the advantages of this method is that we delegate many of the intricacies of loading Mach-O binaries to macOS. But what about if we wanted to stay clear of messing with dyld, and instead roll-up our sleeves and build our own loader? How does all of this byte mapping actually work?

In this blog we'll look at what it takes to construct an in-memory loader for Mach-O bundles within MacOS Ventura without using dyld. We'll walk through the lower-level details of what makes up a Mach-O file, how dyld processes load commands to map areas into memory, and how we can emulate this to avoid writing payloads to disk. I also recommend reading this post alongside the code published here to fully understand the individual areas called out.

In keeping with Apple's migration to ARM architecture, this post will focus on the AARCH64 version of MacOS Ventura and XCode targeting macOS 12.0 and higher. With that said, let's dig in.

What Makes a Mach-O File?

To begin we'll need to understand the layout of a Mach-O file. One of the best resources out there to help us to get our head around the many bits of a Mach-O container is the Mach-O File Format Reference from Aidan Steele which I recommend reviewing.

As we're dealing with the ARM version of MacOS, we will assume that the Mach-O that we're looking at isn't encapsulated in the Universal 2 format (or has been lipo'd), so the first thing that we'll encounter at the beginning of the file is the `mach_header_64`:

```
struct mach_header_64 {
    uint32_t    magic;
    cpu_type_t  cputype;
    cpu_subtype_t    cpusubtype;
    uint32_t    filetype;
    uint32_t    ncmds;
    uint32_t    sizeofcmds;
    uint32_t    flags;
    uint32_t    reserved;
};
```

To construct our loader, we'll need to sanity check a few of these fields:

- `magic` - This field should hold a value of `MH_MAGIC_64`.
- `cputype` - For M1, this should be `CPU_TYPE_ARM64`.
- `filetype` - We're going to be checking for `MH_BUNDLE` type for this post but loading different types should be easy as well.

Once we're happy that the Mach-O is sane, we move onto processing the load commands which immediately follow the `mach_header_64` struct.

Load Commands

A load command, as the name suggests, is a data structure used to instruct dyld on how it should load an area of the Mach-O.

Each load command is represented by the `load_command` struct:

```
struct load_command {
    unsigned long cmd;
    unsigned long cmdsize;
};
```

The `cmd` field ultimately determines what the `load_command` actually represents. For example, let's take a very simple `load_command` of `LC_UUID` which is used to associate a UUID with the binary. This has the layout:

```
struct uuid_command {
   uint32_t cmd;
   uint32_t cmdsize;
   uint8_t uuid[16];
};
```

As you can see, this overlaps with the `load_command` struct, which is why we have the matching fields. This is the case for the various load command supported as we'll see.

Mach-O Segments

One of the first `load_command`'s that we're going to deal with when loading a Mach-O is `LC_SEGMENT_64`.

The segment command tells dyld how to map an area of the Mach-O into virtual memory, what size it should be, what protection is should have, and where the contents of the file are. Let's look at its structure:

```
struct segment_command_64 {
    uint32_t    cmd;
    uint32_t    cmdsize;
    char        segname[16];
    uint64_t    vmaddr;
    uint64_t    vmsize;
    uint64_t    fileoff;
    uint64_t    filesize;
    vm_prot_t   maxprot;
    vm_prot_t   initprot;
    uint32_t    nsects;
    uint32_t    flags;
};
```

For our purposes, we're going to be paying attention to:

- `segname` - The name of the segment, for example, `__TEXT`.
- `vmaddr` - The virtual address where the segment should be loaded. For example, if this is set to `0x4000`, then we'd load the segment at the allocated base of memory + `0x4000`.
- `vmsize` - The size of virtual memory to be allocated.
- `fileoff` - The offset from the beginning of the file to the contents of the Mach-O that should be copied to virtual memory.
- `filesize` - The number of bytes to copy from the file.
- `maxprot` - The maximum memory protection value that should be assigned to the region of virtual memory.
- `initprot` - The initial memory protection that should be assigned to the region of virtual memory.
- `nsects` - The number of sections which will follow this segment structure.

At this point we should note that while dyld relies on `mmap` to pull in segments of a Mach-O into memory, if our initial process is executing as a hardened process (and doesn't have something like `com.apple.security.cs.allow-unsigned-executable-memory` in the entitlements), using `mmap` isn't going to be possible unless the bundle we provide is signed using the same developer certificate as the surrogate app. Also, we're trying to build a memory loader, so pulling in the binary from disk in this case wouldn't make much sense.

To work around this, in our POC we will allocate our blob of memory up front and copy it over, for example:

```
vm_allocate(mach_task_self(), (vm_address_t*)&baseAlloc, maxVirtMemSize, VM
if (baseAlloc == NULL) {
    printf("[!] Error allocating %llx bytes of memory\n", maxVirtMemSize);
    return;
}
```

As with our dyld post previously, we will need to use the correct entitlements in the host binary to allow unsigned executable memory.

Sections

So, as you can see from the fields above, another reference exists within a segment load command, and that's a section.

As the section resides within a segment, while it will inherit its memory protection, it has its own size and file content to be loaded. The data structure for each segment is appended to the segment command and its structure is:

```
struct section_64 {
    char        sectname[16];
    char        segname[16];
    uint64_t    addr;
    uint64_t    size;
    uint32_t    offset;
    uint32_t    align;
    uint32_t    reloff;
    uint32_t    nreloc;
    uint32_t    flags;
    uint32_t    reserved1;
    uint32_t    reserved2;
    uint32_t    reserved3;
};
```

Again, we'll just focus on a few of these fields which are useful for our immediate purpose of constructing a loader:

- `sectname` - The name of the section, for example, `__text`.
- `segname` - The name of the segment associated with this section.
- `addr` - The virtual address offset to be used for this section.
- `size` - The size of the section in the file (and in virtual memory).
- `offset` - The offset to the contents of the section in the Mach-O file.
- `flags` - Flags can be assigned to a section which help determine the values in `reserved1`,`reserved2` and `reserved3`.

As we've already allocated each segment, our loader will just walk through each section descriptor, ensuring that the correct file content is copied into virtual memory.

We need to note here that memory protection may need to be updated as we copy. MacOS for ARM does not allow Read/Write/Execute pages of memory (unless the `com.apple.security.cs.allow-jit` entitlement is used alongside `MAP_JIT`), so we need to accommodate this as we copy:

```
sectionLoadAddr = VMADDR(section->addr);

// Update the memory protection so we can copy over the data for this secti
ret = vm_protect(mach_task_self(), (vm_address_t)sectionLoadAddr, section->
if (ret != 0) {
    printf("\t[!] Error during vm_protect: %d\n", ret);
```

```
}

// Copy the data
memcpy(sectionLoadAddr, base + section->offset, section->size);

// Reset memory protection to the segment
ret = vm_protect(mach_task_self(), (vm_address_t)sectionLoadAddr, section->
if (ret != 0) {
    printf("\t[!] Error during vm_protect: %d\n", ret);
}
```

Symbols

With our loader starting to take shape, we next need to look at how symbols are handled.

Symbols play an important role in the loading process of Mach-O binaries, associating names and ordinals to areas of memory for us to reference later.

Symbols are handled via a load command of `LC_SYMTAB`, which look like this:

```
struct symtab_command {
    unsigned long    cmd;
    unsigned long    cmdsize;
    unsigned long    symoff;
    unsigned long    nsyms;
    unsigned long    stroff;
    unsigned long    strsize;
};
```

Again, we'll focus on the fields we need for constructing a loader:

- `symoff` - Offset from start of the file to an array of `nlist` structures containing information on each symbol.
- `nsyms` - The number of symbols (or `nlist` structures).
- `stroff` - The file offset to the strings used by the symbol lookup.

Obviously to follow along with this we're going to need to know what an `nlist` is:

```
struct nlist_64 {
   union {
     uint32_t n_strx;
   } n_un;
   uint8_t  n_type;
   uint8_t  n_sect;
   uint16_t n_desc;
   uint64_t n_value;
};
```

This structure gives us information about a named symbol:

- `n_strx` - Offset from the symbol strings field to the string of this symbol.
- `n_value` - Contains the value of the symbol, such as the address.

As we will need to reference symbols later, our loader needs to store this information for later:

```
symbols[@(stringTable + nl[i].n_un.n_strx)] = [NSValue valueWithPointer:(vo
```

Dylib's

Next up we have the `LC_LOAD_DYLIB` load command, which references external dylib's to be loaded at runtime.

```
struct dylib_command {
    unsigned long    cmd;
    unsigned long    cmdsize;
    struct dylib    dylib;
};
```

The entry that we need from this is found in the `dylib` struct member, specifically `dylib.name.offset` which is an offset from the beginning of this load command to a string containing the dylib to load.

We'll require this information later on when it comes to relocations, with the order in which dylib's were imported playing an important role, so we'll build an array of dylib's that are referenced for later use:

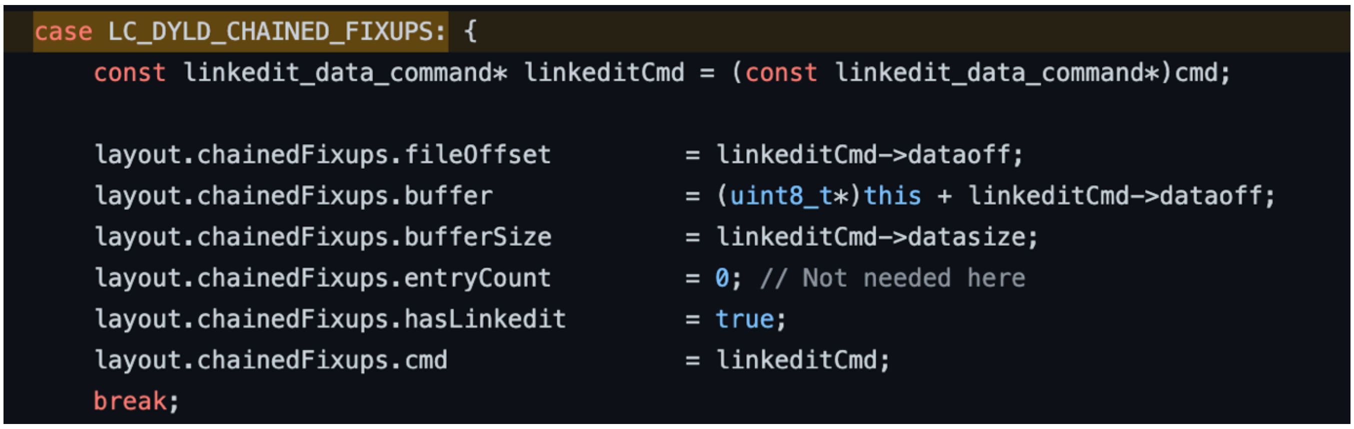
```
char *dyldName = (char *)dylib + dylib->dylib.name.offset;
[dylds addObject:@(dyldName)];
```

Relocations

Now we move onto the more complicated part of Mach-O.. relocations.

Mach-O's built with XCode targeting macOS 12.0 and higher use a load command of `LC_DYLD_CHAINED_FIXUPS`. There isn't much documentation on how this all works, but one of the better resources out there was from Noah Martin reviewing lookup chains for iOS 15. We can also find details on the structs used in Apple's XNU repo here.

Dyld's source shows us that this load command starts with a `struct linkedit_data_command`:



```
case LC_DYLD_CHAINED_FIXUPS: {
    const linkedit_data_command* linkeditCmd = (const linkedit_data_command*)cmd;

    layout.chainedFixups.fileOffset         = linkeditCmd->dataoff;
    layout.chainedFixups.buffer             = (uint8_t*)this + linkeditCmd->dataoff;
    layout.chainedFixups.bufferSize         = linkeditCmd->datasize;
    layout.chainedFixups.entryCount         = 0; // Not needed here
    layout.chainedFixups.hasLinkedit        = true;
    layout.chainedFixups.cmd                = linkeditCmd;
    break;
```

Using the `dataoff` we get to the header:

```
struct dyld_chained_fixups_header
{
    uint32_t    fixups_version;
    uint32_t    starts_offset;
    uint32_t    imports_offset;
    uint32_t    symbols_offset;
    uint32_t    imports_count;
    uint32_t    imports_format;
    uint32_t    symbols_format;
};
```

The first thing we need to do is to gather all imports and construct an ordered array that we will later reference. To do this we'll use the fields:

- `symbols_offset` - Offset from the start of this struct to the symbol strings used by the imports.
- `imports_count` - Number of import entries.
- `imports_format` - The format of any imported symbols.
- `imports_offset` - Offset from the start of this struct to the import table.

Each import entry data structure depends on the `imports_format` field, but typically I've seen this to be the `DYLD_CHAINED_IMPORT` format:

```
// DYLD_CHAINED_IMPORT
struct dyld_chained_import
{
    uint32_t    lib_ordinal :  8,
                weak_import :  1,
                name_offset : 23;
};
```

So essentially this is an array of 32-bit entries. Here we have the `lib_ordinal` field, which is an index into the ordered `dylib` array that we constructed earlier from the `LC_LOAD_DYLIB` load command. The index starts at `1` rather than `0`, meaning that the first index is 1, then 2 etc:

```
const char *dyldName = [dylds[ordinal-1] UTF8String];
```

If the index value is either `0` or `253`, then the entry is referencing `this-image` (the current executing binary). This is the reason we constructed our symbol dictionary earlier, as now we can simply resolve the referenced symbol name in our own binary to its address:

```
if (ordinal == 253 || ordinal == 0) {
        // this-image
        printf("\t[*] Library name: this-image\n");
        func = [symbols[@(symbolNames + chainedImports[i].name_offset)] poi
        func = VMADDR((unsigned long long)func);
}
```

The `name_offset` is an offset into the `symbols_offset` strings that we gathered from `dyld_chained_fixups_header`.

Using this information, we need to build up an ordered array of imports, as we'll need to reference this ordered array in a moment.

With a list of imports constructed, we move onto chained starts, which can be found from the `starts_offset` header field of the `dyld_chained_fixups_header` struct.

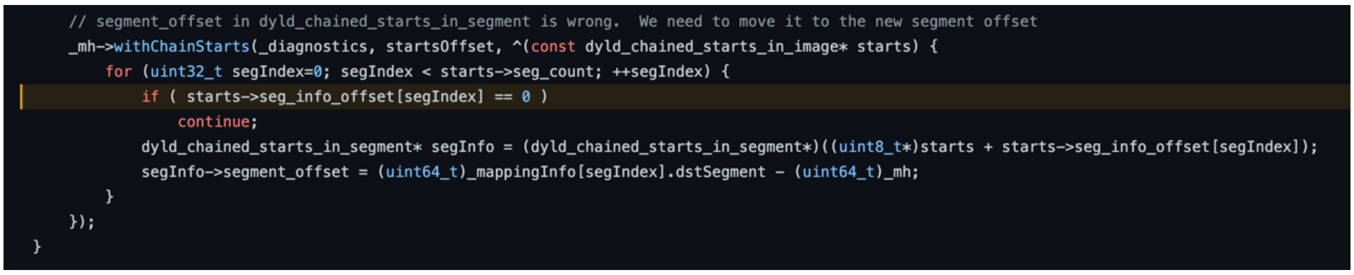
The structure of the chained starts is:

```
struct dyld_chained_starts_in_image
{
    uint32_t    seg_count;
    uint32_t    seg_info_offset[1];  // each entry is offset into this stru
    // followed by pool of dyld_chain_starts_in_segment data
};
```

To navigate this, we'll need to iterate through each entry in `seg_info_offset`, which gives us a list of pointers to `dyld_chained_starts_in_segment`:

```
struct dyld_chained_starts_in_segment
{
    uint32_t    size;               // size of this (amount kernel needs to
    uint16_t    page_size;          // 0x1000 or 0x4000
    uint16_t    pointer_format;     // DYLD_CHAINED_PTR_*
    uint64_t    segment_offset;     // offset in memory to start of segment
    uint32_t    max_valid_pointer;  // for 32-bit OS, any value beyond this
    uint16_t    page_count;         // how many pages are in array
    uint16_t    page_start[1];      // each entry is offset in each page of
                                    // or DYLD_CHAINED_PTR_START_NONE if no
 // uint16_t    chain_starts[1];    // some 32-bit formats may require mult
                                    // for those, if high bit is set in pag
                                    // is index into chain_starts[] which i
                                    // the last of which has the high bit s
};
```

First a note about this struct.. sometimes `segment_offset` is `0`... no idea why... and it looks like [dyld identifies] this too and just ignores them, so we'll do the same :/



```
    // segment_offset in dyld_chained_starts_in_segment is wrong.  We need to move it to the new segment offset
    _mh->withChainStarts(_diagnostics, startsOffset, ^(const dyld_chained_starts_in_image* starts) {
        for (uint32_t segIndex=0; segIndex < starts->seg_count; ++segIndex) {
            if ( starts->seg_info_offset[segIndex] == 0 )
                continue;
            dyld_chained_starts_in_segment* segInfo = (dyld_chained_starts_in_segment*)((uint8_t*)starts + starts->seg_info_offset[segIndex]);
            segInfo->segment_offset = (uint64_t)_mappingInfo[segIndex].dstSegment - (uint64_t)_mh;
        }
    });
}
```

The fields that we need to find the start of each reloc chain to follow will be:

- `pointer_format` - The type of `DYLD_CHAINED_PTR_` struct that the chain uses.
- `segment_offset` - The absolute offset in memory of the segment start address.
- `page_count` - The number of pages that are in the `page_start` member array.
- `page_start` - The offset from the page to the chain start.

When we have a valid offset into a segment, we can start following the reloc chain. Iterating over each entry, we need to check the first bit to determine if the entry is a `rebase` (set to 0) or a `bind` (set to 1):

In the case of a rebase, the entry is cast to a `dyld_chained_ptr_64_rebase` and we update the entry using the `target` offset to the base of our allocated memory.

```
struct dyld_chained_ptr_64_rebase
{
    uint64_t    target    : 36,    // 64GB max image size (DYLD_CHAINED_PTR
                high8     :  8,    // top 8 bits set to this (DYLD_CHAINED_
                reserved  :  7,    // all zeros
                next      : 12,    // 4-byte stride
                bind      :  1;    // == 0
};
```

In the case of a bind, we use a `dyld_chained_ptr_64_bind` and the `ordinal` field is the offset into the `import` array that we constructed earlier.

```
// DYLD_CHAINED_PTR_64
struct dyld_chained_ptr_64_bind
{
    uint64_t    ordinal   : 24,
                addend    :  8,   // 0 thru 255
                reserved  : 19,   // all zeros
                next      : 12,   // 4-byte stride
                bind      :  1;   // == 1
};
```

Then, we need to move onto the next bind or rebase, which is done by doing `next` * 4 (4 bytes is the stride size). We repeat this until the `next` field is `0`, indicating that the chain has ended.

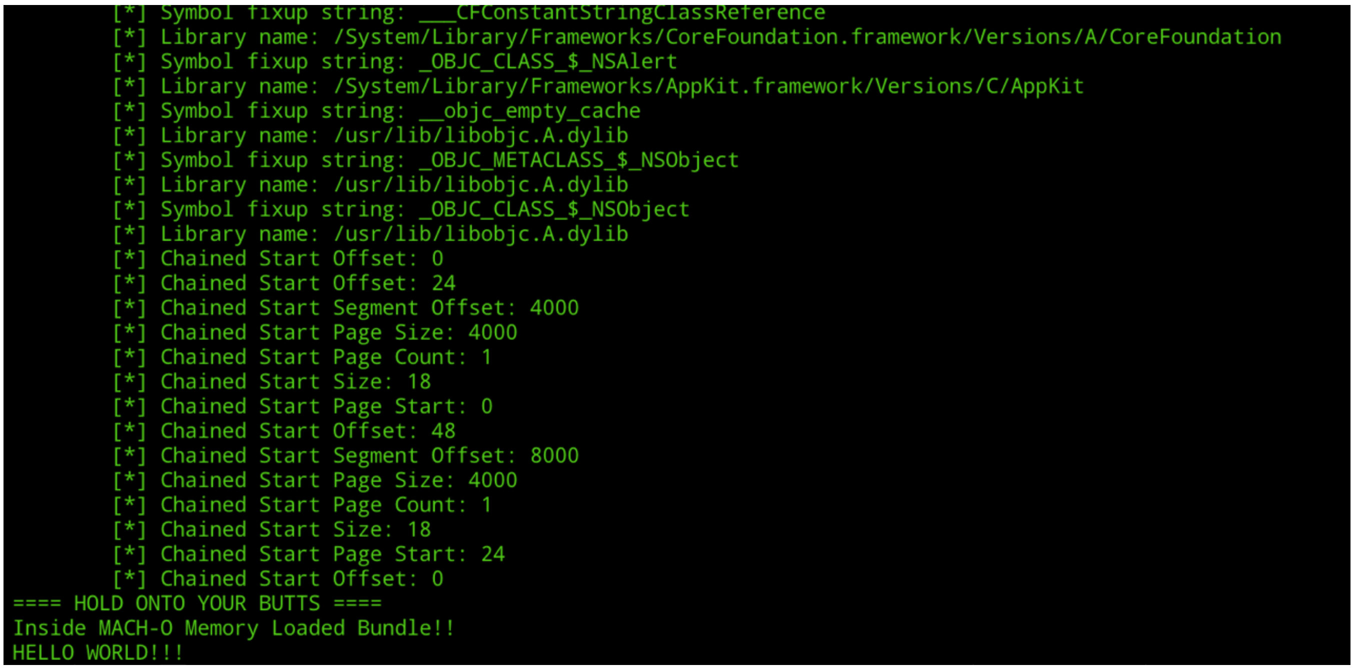
Putting Everything Together

Now we have all the parts, we need to actually construct the loader from start to finish. To recap, this will be:

1. Allocate a region of memory.
2. Load each segment into virtual memory based on the `LC_SEGMENT_64` command.
3. Load each section into each segment.

4. Build an ordered collection of dylib's from the `LC_LOAD_DYLIB` command.
5. Build a collection of symbols from the `LC_SYMTAB` command.
6. Walk the `LC_DYLD_CHAINED_FIXUPS` chain and rebase/bind each reloc.

Once done, we can then use the data from `LC_SYMTAB` to reference the symbol that we want to enter as and pass execution over. If everything goes well, we'll see that our Mach-O is loaded in memory and execution starts :)



```
        [*] Symbol fixup string: ___CFConstantStringClassReference
        [*] Library name: /System/Library/Frameworks/CoreFoundation.framework/Versions/A/CoreFoundation
        [*] Symbol fixup string: _OBJC_CLASS_$_NSAlert
        [*] Library name: /System/Library/Frameworks/AppKit.framework/Versions/C/AppKit
        [*] Symbol fixup string: __objc_empty_cache
        [*] Library name: /usr/lib/libobjc.A.dylib
        [*] Symbol fixup string: _OBJC_METACLASS_$_NSObject
        [*] Library name: /usr/lib/libobjc.A.dylib
        [*] Symbol fixup string: _OBJC_CLASS_$_NSObject
        [*] Library name: /usr/lib/libobjc.A.dylib
        [*] Chained Start Offset: 0
        [*] Chained Start Offset: 24
        [*] Chained Start Segment Offset: 4000
        [*] Chained Start Page Size: 4000
        [*] Chained Start Page Count: 1
        [*] Chained Start Size: 18
        [*] Chained Start Page Start: 0
        [*] Chained Start Offset: 48
        [*] Chained Start Segment Offset: 8000
        [*] Chained Start Page Size: 4000
        [*] Chained Start Page Count: 1
        [*] Chained Start Size: 18
        [*] Chained Start Page Start: 24
        [*] Chained Start Offset: 0
==== HOLD ONTO YOUR BUTTS ====
Inside MACH-O Memory Loaded Bundle!!
HELLO WORLD!!!
```

All code for this POC has been added to the Dyld-DeNeuralyzer project [here](https://blog.xpnsec.com/building-a-mach-o-memory-loader-part-1/).

Next Time...

So I think that's a good place to stop for now, as we've covered quite a lot. One thing you'll quickly see however is that while you can load a C/C++ bundle just fine, if you do try and load an Objective-C bundle, you'll be greeted with something like this:

```
libc++abi: terminating with uncaught exception of type NSException
*** Terminating app due to uncaught exception 'NSInvalidArgumentException', reason:
    '-[NSAlert setMessageText:]: unrecognized selector sent to instance 0x102204080'
terminating with uncaught exception of type NSException
Program ended with exit code: 9
```

This is because of something that happens within dyld when Objective-C Mach-O's are loaded. We'll cover how to implement this the final part in this series.

Standing On The Shoulders...

There are plenty of resources out there which can help with constructing a loader, but the following were particularly useful in helping me to get my head around this. Thank you to all the authors for sharing their knowledge!

- Reference on the Mach-O file format - https://github.com/aidansteele/osx-abi-macho-file-format-reference
- Information on dyld reloc chains - https://opensource.apple.com/source/xnu/xnu-7195.81.3/EXTERNAL_HEADERS/mach-o/fixup-chains.h.auto.html
- Brilliant blog post on why dyld reloc chains are now used - https://www.emergetools.com/blog/posts/iOS15LaunchTime