Literature Review

Harry Moulton — SRN: 18023751

November 2022

1 Introduction

My project, named HTool, will attempt to solve the problem posed by the lack of available tools for static analysis, parsing and reverse engineering binaries and firmware files on Apple's iOS, macOS and derived Operating Systems.

This heavily relates to my chosen field of study, Software Engineering, as the resulting artefact will be a complex application that implements numerous algorithms for handling a range of firmware files that use, in some cases, undocumented file formats and structures.

There is currently a distinct lack of available tools and applications for this purpose, particularly for beginners who want to explore the internals of these operating systems. Two of the most popular tools, JTool (**levin-jtool**) and IDA64 (**hexrays-ida64-pro**), are either no longer regularly updated or extremely expensive. With the exception of some highly specific tools for particular functions available on Github, there are no open-source or recent alternatives to JTool.

In this essay I aim to cover the four primary areas that my project relates to: Mach-O files, iOS Kernel Cache files, iBoot and macOS firmware files. I'll discuss the available literature, the knowledge I have gained from them, and how they have influenced my initial project idea. Due to the iOS & macOS community being fairly small, there is also a lack of research papers - most of the knowledge is shared on blogs, forums and Twitter. In these cases I have outlined the author and why I believe them to be a reputable source.

2 Themes

2.1 Mach-O File Format

Parsing Mach-O files will be a primary requirement of this project. All iOS and macOS applications are compiled as Mach-O executables as well as the XNU kernel. Essentially any executable code with the exception of the boot loader use the Mach-O format. This means that handling this file format is the first functionality that would need to be implemented into any solution.

In 2020 I developed an open-source library called The Libhelper Project (**h3adsh0tzz-libhelper**) written in C. This library provided APIs for working with strings, lists, compression, file handling, Apple's Image4 format and Mach-O files. With regards to Mach-O files, the library takes a given file and translates this into a C structure - however it does not do any further analysis of the files. To make use of the parsed file an application, such as HTool, would build on top of this API to provide the required functionality to match that of existing solutions.

An article published by Scott Lester, Cyber Security Director at Red Maple Technologies, analyses Mach-O files in significant detail. The article "A look at Apple Executable Files" (**lester-macho**) covers the basis of Mach-O files like the header, load commands and segment commands, however he also explores more complex aspects such as how code-signing is embedded in special segment commands - something libhelper does not support and would need to be implemented in HTool.

I briefly cover the basics of the Mach-O format in an article I wrote on my blog entitled "Mach-O File Format: Introduction" (**moulton-macho**). I also cover the Mach-O header, load commands and segment commands, and how these can be handled in C. However, the article doesn't act as a user guide for libhelper, nor does it cover some other aspects of Mach-O files such as the different types, code signing and entitlements as it was only meant to be a quick-to-read introduction.

Lester's article is more useful in the context of the problem as it covers both the Mach-O format in far greater detail than anyone with exception of Apple's developer documentation, and discusses aspects such as code signing and entitlements - this is especially important information as code-signing is something I am not overly familiar with but will need to implement support for in HTool.

While my own article does introduce libhelper, which will be extensively used in HTool, I do not go into sufficient detail in the article regarding the file format.

My findings from Lester's article have influenced my initial solution as I now have a better understanding of code-signing and entitlements in the context of Mach-O files, and therefore have a basis to implement support for these into the solution. As JTool already has support for these it will be an important feature to ensure HTool can match JTool's functionality.

2.2 Kernel Cache

The "Kernel Cache" is a firmware file found on iOS, and now macOS with the introduction of the T2 security co-processor and ARM-based CPUs. (**esser-hackers-handbook** page 249, **levin-os-internals-vol3** page 470) The kernel cache is a combination of the XNU kernel binary and a collection of "Kernel Extensions". These kernel extensions are typically used for device-specific functionality, like storage or power management, with most common code found in the XNU kernel. (**esser-hackers-handbook**, page 249)

The Kernel (including kernel extensions/drivers) is the source of most vulnerabilities on Apple platforms, as can be seen from the iOS 16 Security Updates (**apple-ios16-security-content**). Therefore, support and understanding of the kernel format is vital for this project.

Where Mach-O's have extensive public documentation, there is a lack of in-depth research into the format of the kernel. There are two possible reasons for this, the first being Apple's commitment to "Security through Obscurity" (**security-through-obscurity**), the belief that if they keep as much of the platform a secret it is less likely that security vulnerabilities will be found. The second being the fact the format is tweaked every few versions of iOS.

The topic of the caches format has not been something that has been the centre of attention, instead it is typically a side note in some related research.

Brandon Azad, formerly of Google's Project Zero security research team, published in 2018 an article entitled "Analysing the iOS 12 kernel caches tagged pointers" (**azad-tagged-pointers**). The primary focus of the article was to explore the introduction of the ARMv8.3 architecture and its associated Pointer Authentication (**rutland-pac-slides**) extension, or PAC. A side note of his research was that he noticed a change in the format of the kernel cache between iOS 12 and iOS 11, particularly that certain segments such as __TEXT and __TEXT_EXEC, which hold executable code, are now larger, whereas __PRELINK_INFO was missing some XML data that was used as a map to determine where KEXTs resided in the cache.

Azad observed that: "There appear to be at least 3 distinct kernel cache formats" (**azad-tagged-pointers**). The three formats being:

1. iOS 11: Format used on iOS 10 and 11. It uses a split-kext style, untagged pointers and has a few thousand symbols.
2. iOS 12-normal: Format used on the iOS 12 beta for iPhone9,1. It is similar to iOS 11 but with some structural changes that confuse existing analysis applications.
3. iOS 12-merged: Format used on iOS 12 beta for iPhone 7,1. It is missing prelink segments, KEXTs are merged (meaning all KEXT __TEXT segments are together, __DATA, etc), uses the new tagged pointers and has no symbols.

Along with support for the various different formats of the kernel cache, HTool ideally should able to detect any known and identifiable security mitigations. One example would be pointer authentication. Brandon Azad authored an additional article on pointer authentication "Examining Pointer Authentication on the iPhone XS", focusing more specifically on the iPhone XS (**azad-pac-indepth**).

Azad demonstrated a script for detecting the different types of pointer tagging used, and which areas of the code they were used. This concept could possibly be a useful feature of HTool. He also created a formula for calculating where a pointer starts relative to a tag, where **P** is the tagged pointer, and **A** is the address of that tagged pointer:

```
A + ((P >> 49) & ~0x3)
```

A further example would be Kernel Patch Protection. Sijun Chu and Hao Wu cover this in their paper "Research on Offense and Defense Technology for iOS Kernel Security Mechanism" (**sijun-kernel-paper**) published in 2018. Security researcher Xerub, of Dataflow Security, also authored an in-depth technical writeup of how KPP works (**xerub-tick-tock**). Although this is a security mechanism that is no longer used in modern versions of iOS, it is still a goal to have the project support

analysis of different security mechanisms across a range of iOS versions.

The two articles written by Brandon Azad (**azad-tagged-pointers**, **azad-pac-indepth**) were useful as he details three existing kernel cache formats and the versions they were used in, as well as some technical details of Apple's implementation of Pointer Authentication.

Unfortunately there is currently no documentation on the other Kernel Cache formats, leaving pre-iOS 10 and post-iOS 12 formats for me to investigate myself as part of this project. The "merged style" format (**azad-tagged-pointers**) that Azad mentions is introduced in iOS 12 has been replaced with a new "fileset style" in iOS 15, and further tweaked in iOS 16. The initial development of HTool as a Mach-O parser would be useful here, as analysing segments such as __PRELINK_INFO where KEXT information is kept is required to understand any changes made between versions.

Chu and Wu's paper, along with Xerub's article, were also useful for understanding Kernel Patch Protection. Xerub's writeup in particular is helpful in gaining an understanding as to whether implementing an algorithm for detecting KPP is both useful and feasible - something that will require further practical research, i.e. reverse engineering iOS 9 Kernel's myself.

2.3 iBoot

iBoot is the collective name given to the boot loader components of Apple's iOS and macOS devices. It is made up collectively of the SecureROM, LLB, iBSS, iBEC and iBoot (or iBootStage2). These components serve different purposes; iBSS runs when the device is being updated, iBEC when it fails to boot, and LLB has been made redundant in recent versions. iBootStage2, or as its primarily known "iBoot" is the main boot loader. (**levin-os-internals-vol2-iboot**)

The SecureROM is the most secure component of an iOS/macOS device. It's a piece of software that is burned into the CPU, or Application Processor as Apple refers to it, during manufacturing. It cannot be modified. This means that if a security vulnerability is found it is extremely powerful and cannot be fixed without a CPU design revision. (**esser-hackers-handbook**, Page 300)

Bugs in the SecureROM and iBoot are particularly sought after due to the significant control they have over the device. Some SecureROM binaries have been posted online, primarily on the website securerom.fun, and iBoot binaries are regularly decrypted and posted online too. Therefore, supporting these binaries is required. iBoot exploits, while patchable, are almost as powerful as SecureROM exploits. (**esser-hackers-handbook**, Page 300-301)

A security researcher known only as "B1n4r1b01" noticed that iBoot has several embedded firmwares as well as the actual iBoot binary. There is no well-defined structure like there is with the Kernel Cache, instead it appears the firmwares are just appended one after the other. In a Tweet posted by B1n4r1b01, he demonstrates a tool he developed that can detect the embedded firmwares in a given iBoot binary (**binaryboy-iboot-tweet**).

He adds some further details on the Github page for the tool - Rasegen (**rasegen-github**). He documents the different firmwares, such as storage and power management, as well as the devices that they are found on. However, this tool has not been updated in a few years and it is possible that the format has changed or additional firmwares have been added.

Jonathan Levin, a security researcher who previously focused on iOS and then moved onto Android, also the author of JTool, has written multiple books on the topic of Apple platform security. In his book *OS Internals: Volume II, Levin documents the entire boot process of an iPhone, from the SecureROM to LLB to iBoot to the Kernel, as well as the threat model of iBoot - particularly the vulnerability of the USB stack where most iBoot and SecureROM exploits have been found. He also discusses iBoot's "Relocation Loop", a clever piece of code where iBoot finds itself in memory and relocates to a specific address. (**levin-os-internals-vol2-iboot**)

The Tweet and Github repository from B1n4r1b01 is helpful in understanding that decrypted iBoot binaries contain multiple firmwares that need to be identified. When it comes to developing the algorithm for analysing iBoot binaries, the open-source code he has released will be a helpful reference and basis for my implementation,

Jonathan Levin's chapter on iBoot, and the book as a whole, provide an excellent reference and source of information for the internals of iOS and macOS. Although he does not cover embedded firmwares as B1n4r1b01 has, nevertheless his extensive research and knowledge on the topic make it a very reliable source, the level of detail isn't matched by anyone else.

2.4 macOS

In the past macOS has typically remained a less restricted and locked-down operating system compared to iOS. Until recently macOS did not have a Kernel Cache, instead the kernel shipped as a single binary part of the Base Image, with extensions also stored as individual binaries on the system partition. They were Mach-O files, just not merged together in a single file like a kernel cache.

There are two types of Mac's that used ARM-based processors. First are the T2 Mac's, and second are the "Apple Silicon" Mac's.

A "T2-enabled Mac" is a macOS device that uses an Intel x86_64 processor as its primary Application Processor, and an ARM-based chip from an older generation iPhone that is used as a security co-processor. On these machines the T2 handles the Secure Boot chain, Apple Pay and Touch ID. (**duo-labs-t2-intro**)

The T2 essentially acts as its own system and handles requests for the main processor. It runs a slimmed-down version of watchOS (which itself is a derivative of iOS), known as BridgeOS. This OS uses a Kernel Cache like on iOS, and therefore a tool designed for analysing firmware files of iOS devices would work just fine with BridgeOS.

The second type is an "Apple Silicon" Mac. These machines use ARM-based processors designed by Apple, like in the iPhone or iPad. There is no need for a T2 chip on these machines, so BridgeOS is not present. As macOS becomes more like iOS, it now also uses Kernel Cache in much the same way.

Linux kernel developer Hector Martin, who is currently heading a project to port the Linux Kernel to the ARM-based Mac's, notes the following about the Kernel cache on macOS:

"The built-in bootloader actually boots iBoot2 from /System/Volumes/Preboot/.../iBoot.img4, and that then loads the Darwin kernel from /System/Volumes/Preboot/.../com.apple.kernelcaches /kernelcache." (**martin-hackernews-macos-kernel**)

From this we can gather than ARM-based Mac's use the same kernel cache as iOS and BridgeOS, and most likely the same format too. Therefore, like BridgeOS, support for the Mac's cache file should be automatic.

On Intel-based Mac's, however, there is again a different format. Intel-based Mac's do not use a Kernel Cache file, rather they store the plain XNU kernel Mach-O file at one directory, /System/Library/Kernels/kernel, and make use of a concept called "Kernel Collections"

A blog run by an individual who goes by the name "hoakley" discussed the differences between x86 and arm-based mac kernel's in an article entitled "Extensions are moving away from the kernel". (**hoakley-extensions-kernel**)

In this article he identifies a number of things. Firstly that there are three types of "Kernel Collections", the "Boot Kext Collection (BKC)", "System Kext Collection (SKC)", and "Auxiliary Kext Collection (AKC)". (**hoakley-extensions-kernel**)

1. The Boot Kext Collection is used on both ARM and x86_64 (Intel) Mac's. On ARM, the KEXTs in the BKC are found in the Kernel Cache, and on x86_64 they are found in the BootKextCollection.kc file under /System/Library/KernelCollections/.

2. The System Kext Collection is only used on x86_64, it is not used on ARM-based models. This is found in the same locations as the BKC, and is named SystemKextCollection.kc.

3. Finally, the Auxiliary Kext Collection is only typically found on x86_64 systems, where it is managed by the kernelmanagerd service. On ARM platforms, it can only be found if the system is placed in low-security mode, otherwise it's ignored completely.

I have gained an understanding of the way kernel extensions work on macOS - something I was unfamiliar of prior to this research. This will be useful when it comes to implementing support for Kernel Collections into HTool. iBoot is the same on all platforms so there isn't much else to learn there.

The findings have given me both the knowledge and confidence to implement support for analysing and reverse engineering the kernel, kernel cache and kernel collections on macOS for both Intel and ARM architectures into HTool.

2.5 Existing Solutions

Existing static analysis & reverse engineering tools		
Tool name	Advantages	Drawbacks
JTool	- Mach-O parser and extensive analysis options - Kernel cache parser - Code signing info parser - Kernel cache kext extracting and kernel decompression - Dyld shared cache parser - ARM64 disassembler (via LLVM API)	- Poorly formatted output - No longer actively developed - Missing handling for firmware files such as iBoot, SEP or Device Tree
OTool	- Extensive Mach-O parser - ARM64 & x86_64 disassembler (via LLVM or OTool API) - Shipped with Xcode	- Extremely muddled and confusing output - Updates depend on macOS version
Objdump	- Feature-rich command line disassembler (ARM64 & x86_64)	- Can only disassemble Object files, no support for working with plain binaries. - Primarily a disassembler, not a Mach-O parser - Updates depend on macOS version
IDA64	- Professional disassembler and decompiler - Interactive GUI - Support for dozens of architectures, including ARM64 and x86_64	- Expensive - Overkill for small tasks such as disassembling a single function

Figure 1: Comparison table of currently available tools for static analysis and reverse engineering of iOS and macOS firmware files.

There are already a few existing applications that provide the required tools for analysing and reverse engineering iOS and macOS firmware files, and they each have their own advantages and drawbacks. This is outlined in **Figure 1**.

My aim is for the HTool project to match the functionality of JTool, which itself absorbs features from `otool(1)`, and other Xcode/LLVM developer tools such as `dyldinfo(1)`, `nm(1)`, `strings(1)`, `codesign(1)` and `llvm-objdump(1)`.

The Xcode developer tools are numerous and often confusing to use. While JTool solved this by combining most of their features into a single tool, the issue remains of often a non-user friendly output. See examples of OTool and JTool output in **Figure 2** and **Figure 3**.

```
$ otool -h -l /bin/ls
Mach header
      magic  cputype cpusubtype  caps    filetype ncmds sizeofcmds      flags
 0xfeedfacf 16777228          2  0x80           2    19       1728 0x00200085
Load command 0
```

```
      cmd LC_SEGMENT_64
  cmdsize 72
  segname __PAGEZERO
   vmaddr 0x0000000000000000
   vmsize 0x0000000100000000
  fileoff 0
 filesize 0
  maxprot 0x00000000
 initprot 0x00000000
   nsects 0
    flags 0x0
Load command 1
      cmd LC_SEGMENT_64
  cmdsize 472
  segname __TEXT
   vmaddr 0x0000000100000000
   vmsize 0x0000000000008000
  fileoff 0
 filesize 32768
  maxprot 0x00000005
 initprot 0x00000005
   nsects 5
    flags 0x0    ...
```

Figure 2: Output of OTool, printing the header (-h) and load commands (-l) of a given binary.

```
$ jtool -h -l /bin/ls
Magic: 64-bit Mach-O
Type: executable
CPU: x86_64
Cmds: 19
size: 1816 bytes
Flags: 0x200085

LC 00: LC_SEGMENT_64          Mem: 0x000000000-0x100000000      __PAGEZERO
LC 01: LC_SEGMENT_64          Mem: 0x100000000-0x100005000      __TEXT
Mem: 0x100004430-0x100004604           __TEXT.__stubs  (Symbol Stubs)
Mem: 0x100004430-0x100004604           __TEXT.__stubs  (Symbol Stubs)
Mem: 0x100004920-0x100004b10           __TEXT.__const
Mem: 0x100004b10-0x100004f66           __TEXT.__cstring        (C-String Literals)
Mem: 0x100004430-0x100004604           __TEXT.__stubs  (Symbol Stubs)
LC 02: LC_SEGMENT_64          Mem: 0x100005000-0x100006000      __DATA
    ...
```

Figure 3: Output of JTool, printing the header (-h) and load commands (-l) of a given binary.

Along with this, JTool does not have any built-in options for handling other firmware files such as SecureROM bins, iBoot, DeviceTree, SEP, SEPOS, and others. There is no version or device detections for binaries, something that would be a useful feature.

2.6 Summary

Throughout this essay I have reviewed a range of sources from a variety of locations, from Tweets and forum posts to technical articles and books. I have covered what I identified at the start as the four main areas the project should focus on - Mach-O files, Kernel Caches, iBoot and macOS.

I have acquired the following knowledge as a result of this essay:

1. There is a distinct lack of resources documenting these firmware file formats, with exception of Mach-O. A potential part of my project that could in part resolve this would be documentation. An in-depth user guide covering how to use the tool, and how the firmware files it can analyse are structured, would fill a knowledge gap that appears to exist - at least with public documentation.

2. I now have a better understanding of how the Kernel cache and KEXTs work on ARM and Intel-based macOS devices. This is something that prior to this essay I did not have prior knowledge of, and this will prove useful when developing my project.

3. I have an understanding of a number of security mitigations, such as Pointer Authentication and Kernel Patch Protection, and how it may be possible to implement algorithms to detect these mitigations in firmware files.

My initial idea was for my project to be a tool that primarily analysed Mach-O and Kernel files, with some support for iBoot and disassembly. However, with the knowledge I know have from this work I believe that this can be expanded, allowing the tool to essentially act as a Swiss army knife for iOS and macOS firmware file analysis.

Based on the knowledge acquired from this essay, I believe the following features are feasible to implement in my project:

1. Mach-O Parser: Header, Load Commands, Segment Commands, Code signing, entitlements.

2. Kernel Analysis: Device & Version detection, Kernel cache format detection, KEXT handling (extract a KEXT to its own file, so it can be treated as a regular Mach-O). Security mitigation detection (KPP, PAC, KTRR)

3. iBoot, SecureROM, etc: Device & Version detection, iBoot embedded firmware handling.

4. Command line disassembly of both Mach-O and non-object files.