Literature Review

Harry Moulton

November 2022

1 Introduction

My project, named HTool, will attempt to solve a problem posed by the lack of available tools for statically analysing, 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 able to handle a range of different firmware file formats and static analysis functionality.

Currently there is a distinct lack of available applications or tools for this purpose. The two main tools, JTool[**jtool**] and IDA64[**ida64**], are either no longer updated or extremely expensive. Apart for highly specific tools available on Github, there is not a great deal of available tools, particularly for people new to security research on Apple platforms.

This essay will cover the primary areas and file formats the project should support, and analyse available research or literature for those areas.

2 Themes

2.1 Mach-O File Format

Parsing Mach-O files is a primary requirement of this project. All iOS and macOS applications are compiled as Mach-O executables and the XNU kernel is also compiled as a Mach-O. Essentially any executable code from the Kernel up will use Mach-O. 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[8] 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 analyses Mach-O files in significant detail. His blog post "A look at Apple Executable Files"[10] 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[5]. I 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 basic 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. 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.

The kernel is the primary source of discovered security vulnerabilities, so 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"[12], 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 analysing a new security feature introduced in iOS 12 entitled "Analysing the iOS 12 kernel caches tagged pointers"[3]. The primary focus of the article was to explore the introduction of the ARMv8.3 architecture and it associated Pointer Authentication[7] extension, or PAC. A side note of his research into PAC was that he noticed the change in format 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 the following: "There appear to be at least 3 distinct kernel cache formats". The three formats he has observed are:

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 different formats of the kernel cache, HTool will need to be able to detect any known and identifiable security mitigations. One example would be pointer authentication. Brandon Azad authored an additional piece of pointer authentication, focusing more specifically on the iPhone XS[4].

However, Azad's first article on PAC demonstrates an interesting concept for an analysis tool - observing the types of pointer tagging, what segments they occur in, and how often. Azad noticed five different types of tagged pointers, and created a formula for calculating where a pointer starts relative to a PAC tag.

Where **P** is a tagged pointer, and **A** is the address of that tagged pointer:

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

Another example would be Kernel Patch Protection. This is covered by Sijun Chu and Hao Wu in their "Research on Offense and Defense Technology for iOS Kernel Security Mechanism"[11] research paper published in 2018, and by well-known security researcher Luca Todesco (aka Qwertyoruiop) in many of his conference presentations. 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.

Azad's research was particularly useful as it details three existing kernel cache formats, leaving only pre-iOS 10 and post-iOS 12 for me to investigate myself. Unfortunately there is no documentation on the kernel cache format introduced in iOS 15. The "merged-style" format that Azad mentions is introduced in iOS 12 has been replaced with a new "fileset-style". Again, this is an area that I would have to investigate myself. The initial development of HTool's Mach-O parser would be useful here as I could analyse the changes in the segments between the different iOS versions.

2.3 iBoot

iBoot is the collective name given to the boot loader components of Apple's iOS and macOS devices[6]. 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.

The SecureROM is the most secretive component on 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 cannot be fixed without a CPU design revision.

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 a website known as securerom.fun, and iBoot binaries are regularly decrypted and posted online too. Therefore, supporting these binaries is required.

As documented by a security researcher known as ”B1n4r1b01”, iBoot has several embedded firmwares as well as the actual iBoot binary. There is not a well-defined structure like there is with the Kernel Cache, instead it appears they're just appended one after the other. B1n4r1b01 demonstrates this in a Tweet where he shows off a tool he developed for detecting these embedded binaries[2].

He adds some further documentation on the Github page for the tool - Rasegen. He details the different firmwares, such as storage and power management, as well as the devices that they are found on. This tool though has not been updated in a few years and it is possible that either the format has changed, or additional firmwares have been added.

Jonathan Levin, another security researcher who previously focused on iOS, and then moved onto Android, and happens to be the author of JTool, has written multiple books on the topic of Apple platform security. His book, ”*OS Internals: Volume II”, covers in detail iBoot [6] , it's function and how it works. Importantly, he discusses the first code that iBoot runs and how it essentially relocates itself in memory.

These two sources have been useful in detailing the inner-workings of iBoot and SecureROM and help form a basis to what will be a feature of HTool that is able to analyse these files, determine the version, device and any other useful information (for iBoot, finding the relocation loop for example).

While B1n4r1b01's research into embedded firmwares in iBoot is useful for understanding the structure of the files, Levin's book's offer a level of detail that isn't matched by anyone else.

2.4 macOS Differences

macOS has typically been very different to that of iOS and other mobile operating systems. It did not have a Kernel Cache, instead it shipped a plain Mach-O Kernel binary, and separate Kernel Extensions split into user and kernel land [1].

However since the introduction of the T2 security co-processor, codenamed iBridge, macOS has gradually moved towards the security model of iOS. Pepijn Bruienne undertook a deep-dive into this new chip in his paper from 2018[9]. iBridge used the same A10 processor as the iPhone 7, and ran a slimmed-down version of watchOS, which Apple named BridgeOS.

Because this new chips primary function was security, and the new Secure Boot architecture on Mac's, this obviously became a target for security researchers. However, because the system is similar to that of watchOS it doesn't require any additional functionality to be analysed - in theory a tool developed for iOS firmware analysis would work just fine for BridgeOS.

Where things begin to change is the introduction of ARM-based ”Apple Silicon” machines. These machines had a vastly different security architecture. No more third party kernel extensions, and macOS now used a Kernel Cache too, helpfully with its own strange proprietary format.

¡ discuss https://news.ycombinator.com/item?id=26113488, find info on macOS kernel collections ¿

2.5 Reverse Engineering

Reverse Engineering is the primary method of finding software bugs and vulnerabilities on Apple platforms. While other techniques are used, such as code auditing, it would be difficult to discover, document and develop a proof-of-concept for a security bug without any reverse engineering.

What does this mean in the context of the HTool project? While this project could never hope to imitate the functionality of applications like IDA64, which are full professional reverse engineering and decompilation tools, HTool needs to be able to match the functionality of both JTool and objdump. What these two applications allow is for the disassembly of provided binaries.

For example, one wants to disassemble the code in an iOS kernel binary between two addresses, a tool named objdump that is shipped with the Xcode Developer Tools can be invoked like so:

```
$ objdump -D kcache-iphone14-5-ios15.arm64
  --start-address=0xfffffff008310128
  --stop-address=0xfffffff008310158

Disassembly of section __TEXT_EXEC,__text:

fffffff007ba4000 <__text>:
fffffff008310128:   adrp x0, 0xfffffff009e14000
fffffff00831012c:   add x0, x0, #0
fffffff008310130:   add x0, x0, x22
fffffff008310134:   sub x0, x0, x23
fffffff008310138:   bl 0xfffffff00831d67c
fffffff00831013c:   mov sp, x0
fffffff008310140:   adrp x0, 0xfffffff009e0c000
fffffff008310144:   add x0, x0, #0
fffffff008310148:   add x0, x0, x22
fffffff00831014c:   sub x0, x0, x23
fffffff008310150:   msr SPSel, #0
fffffff008310154:   mov sp, x0
```

There are two possible options for implementing this kind of functionality. Option 1 would be to develop and implement a completely new and custom disassembly framework, something that is a significant challenge and would require extensive knowledge of the ARM architecture and the ARM reference manual [].

The second option would be to make use of an existing framework such as Capstone Engine.

¡ check and come back to this ¿

References

[1] Apple. *System and Kernel Extensions in macOS*. 2021. URL: `https://support.apple.com/en-gb/guide/deployment/depa5fb8376f/web`.

[2] b1n4r1b01. *A13 iBoots have 4 embedded firmwares*. 2020. URL: `https://twitter.com/b1n4r1b01/status/1237413317901082626`.

[3] Brandon Azad. *Analyzing the iOS 12 kernelcache's tagged pointers*. 2018. URL: `https://bazad.github.io/2018/06/ios-12-kernelcache-tagged-pointers/`.

[4] Brandon Azad. *Examining Pointer Authentication on the iPhone XS*. 2018. URL: `https://googleprojectzero.blogspot.com/2019/02/examining-pointer-authentication-on.html`.

[5] Harry Moulton. *Mach-O File Format: Introduction*. 2020. URL: `https://h3adsh0tzz.com/posts/macho-file-format`.

[6] Jonathan Levin. **OS Internals Volume II*. 2019. URL: `http://newosxbook.com/bonus/iBoot.pdf`.

[7] Mark Rutland. *ARMv8.3 Pointer Authentication*. 2017. URL: `https://events.static.linuxfound.org/sites/events/files/slides/slides_23.pdf`.

[8] Harry Moulton. *The Libhelper Project, Github*. 2020-2022. URL: `https://github.com/h3adshotzz/libhelper/tree/3.0.0`.

[9] Pepijn Bruienne. *Apple iMac Pro and Secure Storage*. 2018. URL: `https://duo.com/blog/apple-imac-pro-and-secure-storage`.

[10] Scott Lester. *A look at Apple executable files*. 2020. URL: `https://redmaple.tech/blogs/macho-files/`.

[11] Sijun Chu Hao Wu. *Research on Offense and Defense Technology for iOS Kernel Security Mechanism*. 2018. URL: `https://aip.scitation.org/doi/pdf/10.1063/1.5033796`.

[12] Wikipedia. *Security through Obscurity*. 2019. URL: `https://en.wikipedia.org/wiki/Security_through_obscurity`.