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 The two main tools, JTool[**jtool**] and IDA64[**ida64**], are either no longer updated or extremely expensive.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[10] 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"[12] 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[6]. 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 one of the firmware files found on an 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 the "Kernel Extensions", or KEXTs, required for each device.[7]

The XNU kernel and its associated KEXTs are the source of the vast majority of iOS and macOS security vulnerabilities. However the file format is proprietary and often depends on the device or iOS version. Different iOS devices and versions also have different security mitigations/features.

2.2.1 File format

Being able to understand the kernel cache is the second most important feature of the project besides understanding Mach-O files. The KEXTs and the kernel binary itself are all Mach-O files, and the actual kernel cache is also a Mach-O - essentially the kernel cache is a collection of Mach-O's embedded in a Mach-O.[**find-reference**].

Where Mach-O's have extensive public documentation, the Kernel Cache does not. There are two possible reasons for this. The first, Apple's commitment to "Security through Obscurity"[14], the belief that if they keep much of the platform a secret its less likely that security vulnerabilities will be found. The second being the fact that the format is tweaked every few iOS versions.

The topic of the caches format is not something that has ever been the centre of attention, instead typically being something that is a small mention in related research.

For example, Brandon Azad - formerly of Google's Project Zero security research team - published in 2018 an article entitled "Analyzing the iOS 12 kernel caches tagged pointers"[4]. The primary focus of this article was the introduction of Pointer Authentication[9] with the ARMv8.3 architecture into that years iPhones. A side note of this research was Azad noticing a significant difference of the kernel cache format between older devices and the new ARMv8.3 devices.

Azad observed the following; "There appear to be at least 3 distinct kernelcache formats".

1. iOS 11-normal: The format used on iOS 10 and iOS 11. It has split kexts, untagged pointers and around 4000 symbols.
2. iOS 12-normal: The format used on iOS 12 beta for iPhone9,1. It is similar to the iOS 11 kernelcache, but with some structural changes that confuse IDA 6.95.
3. iOS 12-merged: The format sed on iOS 12 beta for iPhone7,1. It is missing prelink segments, has merged kexts, uses tagged pointers, and, to the dismay of security researchers, is completely stripped.

This snippet from his research article is particularly useful. For the project to truly be a solution to the problem it needs to, at the very least, support all 64-bit variants of the kernel cache. This snippet gives us a basis for understanding three of these variants.

Kernel caches from iOS versions earlier than iOS 11, and newer than iOS 12 will need further investigation.

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 the new "fileset-style". This is an area I would have to investigate and research during the development of the project. The initial development of HTool's Mach-O parser would significantly help this effort, as I would be able to analyse the different code segments and observe where the KEXTs are placed.

2.2.2 Security features

As well as being able to detect and handle the different formats of Kernel Cache, HTool will also need to be able to detect and handle different security features, as well as disassemble the binaries.

One example would be Pointer Authentication. Brandon Azad has also written an in-depth article on the PAC implementation on iPhone XS[5]. 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"[13] 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.

2.3 iBoot

iBoot is the collective name given to the boot loader components of Apple's iOS and macOS devices. It's made up of the SecureROM, LLB, iBSS, iBEC and iBoot (or iBootStage2). These components serve different purposes, for example iBSS runs when a software updated is being performed and iBEC when the device fails to boot. LLB, short for Low Level Bootloader, used to be a separate component but has since been essentially made redundant. [**find-reference**].

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.

This makes bugs in the SecureROM particularly sought after. However, it is not possible to obtain the SecureROM's code without first either having a vulnerability in the SecureROM itself, or one in iBoot that allow for code execution to dump memory.

This leads us on to iBoot. The SecureROMs job is to load and decrypt iBoot into memory and jump to it, with iBoot then loading and decompressing the kernel. The iBoot firmware image does not just contain iBoot - it contains firmware for other device components such as the storage controller and power management. [3]

As well as this, iBoot is a very complex piece of software. Jonathan Levin mentions in his book "NewOSXBook" how the Relocation Loop works where iBoot "starts with a relocation loop to move the image to a specific virtual address". [8].

Providing one has obtained decryption keys for an iBoot binary being able to determine what embedded firmwares are present and being able to split them up into separate files is a necessary feature of any reverse engineering tool. A security researcher who goes by the name "B1n4r1B01" demonstrated this in a Tweet [2] showing how iBoot has a number of embedded firmwares. He also released this tool on Github with some further documentation [3], however it has not been updated in a few years so it is possible newer versions of iBoot are different.

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[11]. 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] b1n4r1b01. *rasengan - extract various firmware blobs from iBoot*. 2020. URL: https://github.com/b1n4r1b01/rasengan.

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

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

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

[7] iPhoneWiki. *Kernelcache*. 2019. URL: https://www.theiphonewiki.com/wiki/Kernelcache.

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

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

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

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

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

[13] 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.

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