Kextractor - Kernelcache Manipulation Library

Popa Roxana[1] and Răzvan Deaconescu[1]

roxana.popa2703@stud.acs.upb.ro, razvan.deaconescu@upb.ro
[1]Faculty of Automatic Control and Computers, University POLITEHNICA of Bucharest

May 2023

Abstract *Some operating systems, such as iOS, allow third-party software developers to extend the range of their kernel functionality by inserting their own code into kernel mode. This is normally acomplished by using kernel modules, or more commonly known as kernel extensions. These extensions are incredibly powerful as they have full access to kernel space, and therefore they can pose a great security problem. Anyone can inject code directly into kernel space, and there is no way of differentiating between malware and legitimate code. For that reason, a comprehensive analysis of the kernel extensions loaded at runtime can be essential when investigating a potential security incident. Kextractor is an open-source Python library for viewing and extracting kernel extensions from a given iOS kernelcache. It can be used on all versions up to iOS 16 to get a comprehensive list of all extensions from a kernelcache file. Additionally, it is able to both decrypt and decompress all kernelcache versions to provide support for all operations required.*

Keywords – `kernel extension`, `iOS`, `kernelcache`, `reverse engineering`, `security`, `operating systems`

1 Introduction

Apple mobile devices have unquestionably been some of the most used devices, with an estimated 1.06 billion users worldwide. At the base of those emblematic devices is the iPhone operating system, or more commonly known as iOS, and its long list of security features. However, it is practically impossible to build an operating system that offers all the features needed by each device. Therefore, instead of developing dedicated operating systems, Apple has offered third-party developers the possibility of adding custom features directly into the kernel through modules named kernel extensions. These modules are extremely powerful, as they have unbounded access to every resource in kernel space and can be used to define any type of functionality or behaviour. From a security perspective, they pose a great risk as there is no way to distinguish malicious code from legitimate code.

In the event of a security incident, an extensive analysis of the kernel and its extensions could bring crucial information. Also, this kind of analysis has significant educational value and can help one better understand the way that iOS operates.

Kextractor is an open-source tool that extracts all kernel extensions loaded at runtime from a given kernelcache file. Because the kernelcache file is encrypted for iOS versions up to iOS 10 and it is also compressed using various algorithms, it is mandatory for the kernelcache to be decrypted (if needed) and decompressed before any analysis can be performed. To facilitate the process, Kextractor offers support for both decryption and decompression.

2 State of the Art

iOS is the operating system used by all Apple mobile devices. It uses kernel extensions to offer additional features depending on the need of each device. The extensions that need to be loaded at runtime are usually found in the kernelcache file, which is an image of the kernel that will be compiled. As is the case with every piece of code that runs on an Apple device, this file must be signed using a valid certificate. To ensure an additional layer of security, iOS versions up to iOS 10 used an encrypted kernelcache. Moreover, because it is meant to accelerate the boot process, the kernelcache is compressed using Apple proprietary algorithms.

Kernelcache: The kernelcache is an image of the kernel that will be compiled at boot time. It contains all pre-linked extensions loaded by the kernel and it is meant to speed up the boot process. Instead of having the kernel and the extensions load sequentially, the boot loader will load a kernelcache file[1]. This file is usually found in a compressed form: for versions up to iOS 14, the compression algorithm used is LZSS and for later versions LZFSE algorithm is employed. Moreover, versions older than iOS 10 employed encrypted kernelcaches, but as the format of these files changed, encrypting them was deemed unnecessary.

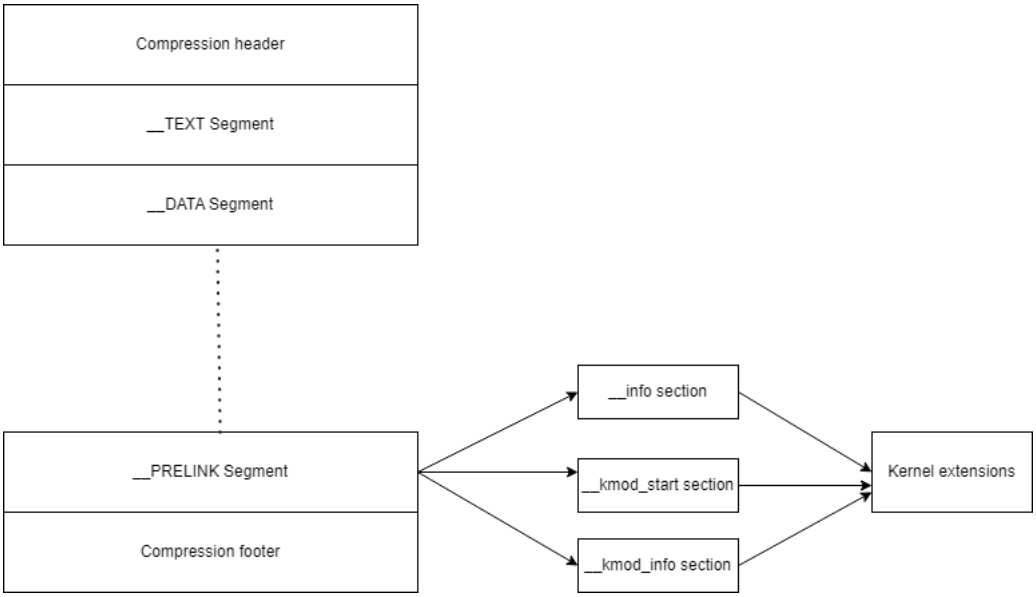

Figure 1: Kernelcache architecture

Code signing: Code signing is a process meant to ensure the authenticity of a code file. Before it can run on an Apple device, every piece of code must be virtually signed with a certificate issued by Apple. Doing so helps guarantee that the code was not modified by untrusted sources. Though it can not protect against malicious code, the certificate used to sign code that was deemed malicious will be revoked, delaying the deployment of the said malware[3].

Security requirements: Because kernel extensions are essentially pieces of code that will be loaded into kernel memory, it is mandatory that they comply to some security requirements. Firstly, the extension must be owned by the root user and should have limited directory permissions. And secondly, any file used by the extension should have only read permissions[1].

Reverse engineering: Reverse engineering is commonly used in security investigations, identifying issues and their root causes. There are currently numerous tools for iOS reverse engineering that operate on different parts of the operating system. Some notable mentions are SandBlaster[5], which is an open-source tool meant for iOS sandbox profile reversing and investigation, and jtool[6], which is a tool meant for general iOS kernel analysis. These are some

representative utilitaries in the iOS reverse engineering community that help investigate and understand the mechanisms of this complex operating system. Since changes related to the kernel are very rarely documented, their use is imperative. Kextractor was designed to contribute to a better comprehension of the iOS operating system.

3 Implementation

This section describes the functionality of Kextractor and the approach we propose for the initial analysis of the kernelcache. Each feature is presented in depth in the following subsections. Finally, a usage example is demonstrated in the last subsection.

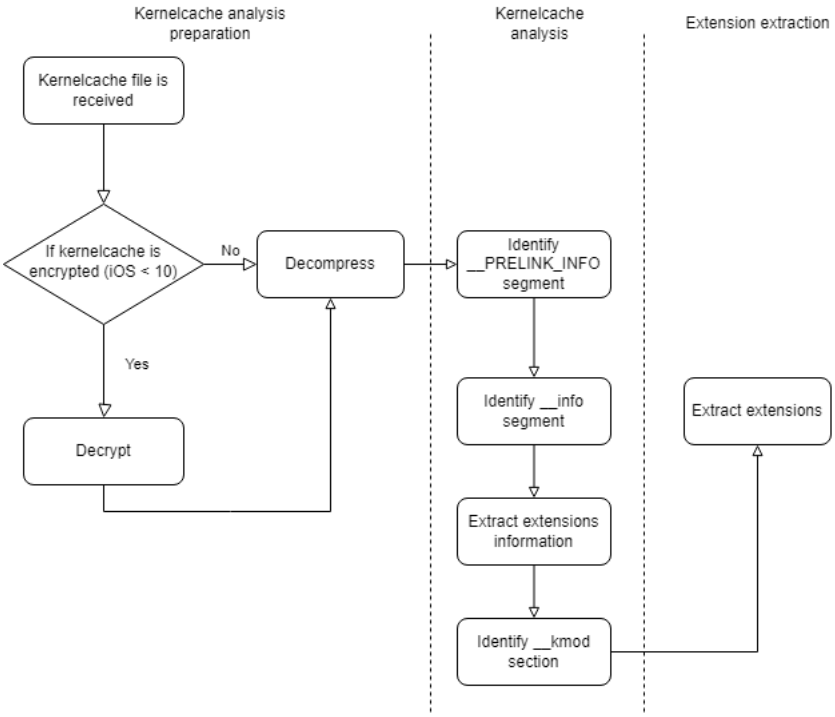

Figure 2: Kextractor functionality overview

3.1 Extension extraction

Kextractor uses an open-source library called **lief** for Mach-O files analysis. It is mainly used for identifying the **__PRELINK_INFO** segment which contains the list of linked extensions and parsing it. The implemenation is based on the following flow:

1. Identifying the __PRELINK_INFO segment.
2. Identifying the __info section in the segment and extracting the information about the extensions.
3. Identifying the __kmod section in the segment and extracting the extensions.

Note that the kernelcache contains numerous segments and subsections, but the actual organization and their role is beyond the scope of this paper.

After identifying the __PRELINK_INFO segment and the __kmod section, the element of interest becomes the **_PrelinkInfoDictionary** dictionary which contains the actual loaded extensions. Because it comes in a XML format, Kextractor performs a series of parsing operations using the **defusedxml** library to obtain the final result.

```
<dict><key>_PrelinkInfoDictionary</key>
 <array>
        <dict>
                <key>CFBundleName</key><string>MAC Framework
                   Pseudoextension</string>
                <key>_PrelinkExecutableLoadAddr</key><integer
                   size="64">0x80346000</integer>
                <key>_PrelinkKmodInfo</key><integer ID="5" size
                   ="32">0x0</integer>
                <key>_PrelinkExecutableSize</key><integer size=
                   "64">0x28c</integer>
                <key>CFBundleDevelopmentRegion</key><string ID=
                   "7">English</string>
                <key>CFBundleVersion</key><string>11.0.0</
                   string>
                <key>_PrelinkExecutableSourceAddr</key><integer
                    size="64">0x80346000</integer>
                <key>CFBundlePackageType</key><string>KEXT</
                   string>
                <key>CFBundleShortVersionString</key><string>
                   11.0.0</string>
                <key>OSBundleCompatibleVersion</key><string>
                   8.0.0b1</string>
        </dict>
        <dict>
                <key>CFBundleName</key><string>Private
                   Pseudoextension</string>
                <key>_PrelinkExecutableLoadAddr</key><integer
                   size="64">0x80347000</integer>
                <key>_PrelinkKmodInfo</key><integer IDREF="5"/>
        </dict>
</array>
</dict>
```

Listing 1: Sample _PrelinkInfoDictionary format

Finally, the output shows the extensions:

```
1 com.apple.iokit.IOSlowAdaptiveClockingFamily (IOSlowAdaptiveClockingFamily) virtaddr=0xffffffff008334988 fileoffset=0x1330988 size=0x1d10
2 com.company.driver.modulename () virtaddr=0xffffffff008336698 fileoffset=0x1332698 size=0x1a1f0
3 com.apple.iokit.IOReporting () virtaddr=0xffffffff008350888 fileoffset=0x134c888 size=0x2988
4 com.apple.driver.AppleARMPlatform (AppleARMPlatform) virtaddr=0xffffffff008353210 fileoffset=0x134f210 size=0x31cd8
5 com.apple.iokit.IONetworkingFamily (I/O Kit Networking Family) virtaddr=0xffffffff008384ee8 fileoffset=0x1380ee8 size=0x1a670
6 com.apple.iokit.IOTimeSyncFamily (IOTimeSyncFamily) virtaddr=0xffffffff00839f558 fileoffset=0x139b558 size=0x18748
7 com.apple.iokit.IOPCIFamily (I/O Kit PCI Family) virtaddr=0xffffffff0083b7ca0 fileoffset=0x13b3ca0 size=0x18af8
8 com.apple.driver.AppleConvergedIPC () virtaddr=0xffffffff0083d0798 fileoffset=0x13cc798 size=0x3f380
9 com.apple.driver.mDNSOffloadUserClient-Embedded (mDNSOffloadUserClient-Embedded) virtaddr=0xffffffff00840fb18 fileoffset=0x140bb18 size=0x2ec8
10 com.apple.iokit.IOSkywalkFamily (IOSkywalkFamily) virtaddr=0xffffffff0084129e0 fileoffset=0x140e9e0 size=0x40890
11 com.apple.driver.AppleIPAppender (AppleIPAppender) virtaddr=0xffffffff008453270 fileoffset=0x144f270 size=0x4280
12 com.apple.driver.AppleConvergedIPCBaseband () virtaddr=0xffffffff0084574f0 fileoffset=0x14534f0 size=0x135b0
13 com.apple.driver.AppleSamsungSPI (AppleSamsungSPI) virtaddr=0xffffffff00846aaa0 fileoffset=0x1466aa0 size=0x54a0
14 com.apple.kec.corecrypto (corecrypto) virtaddr=0xffffffff00846ff40 fileoffset=0x146bf40 size=0x4c878
15 com.apple.kext.CoreTrust (CoreTrust) virtaddr=0xffffffff0084bc7b8 fileoffset=0x14b87b8 size=0x6440
16 com.apple.iokit.IOCryptoAcceleratorFamily (IOCryptoAcceleratorFamily) virtaddr=0xffffffff0084c2bf8 fileoffset=0x14bebf8 size=0x5498
17 com.apple.security.AppleImage4 (AppleImage4) virtaddr=0xffffffff0084c8090 fileoffset=0x14c4090 size=0x111c0
18 com.apple.driver.AppleMobileFileIntegrity (AppleMobileFileIntegrity) virtaddr=0xffffffff0084d9250 fileoffset=0x14d5250 size=0xed88
19 com.apple.iokit.IOHIDFamily (IOHIDFamily) virtaddr=0xffffffff0084e7fd8 fileoffset=0x14e3fd8 size=0x54490
20 com.apple.driver.IOSlaveProcessor (IOSlaveProcessor) virtaddr=0xffffffff00853c468 fileoffset=0x1538468 size=0x1340
21 com.apple.iokit.CoreAnalytics () virtaddr=0xffffffff00853d7a8 fileoffset=0x15397a8 size=0x5818
22 com.apple.driver.AppleA7IOP (AppleA7IOP) virtaddr=0xffffffff008542fc0 fileoffset=0x153efc0 size=0xbf58
23 com.apple.driver.RTBuddy (RTBuddy) virtaddr=0xffffffff00854ef18 fileoffset=0x154af18 size=0x2dcf8
24 com.apple.driver.AppleFirmwareUpdateKext (AppleFirmwareUpdateKext) virtaddr=0xffffffff00857cc10 fileoffset=0x1578c10 size=0x1cf0
25 com.apple.drivers.AppleSPU () virtaddr=0xffffffff00857e900 fileoffset=0x157a900 size=0x33d80
26 com.apple.driver.AppleEmbeddedLightSensor (AppleEmbeddedLightSensor) virtaddr=0xffffffff0085b2680 fileoffset=0x15ae680 size=0x167f8
27 com.apple.driver.AppleS5L8920XPWM (AppleS5L8920XPWM) virtaddr=0xffffffff0085c8e78 fileoffset=0x15c4e78 size=0x1bf8
28 com.apple.driver.AppleBluetoothDebugService (AppleBluetoothDebugService) virtaddr=0xffffffff0085caa70 fileoffset=0x15c6a70 size=0x278
29 com.apple.driver.corecapture (corecapture) virtaddr=0xffffffff0085cace8 fileoffset=0x15c6ce8 size=0x1a908
30 com.apple.driver.AppleBluetoothDebug (AppleBluetoothDebug) virtaddr=0xffffffff0085e55f0 fileoffset=0x15e15f0 size=0xa2c8
31 com.apple.driver.AppleInputDeviceSupport (AppleInputDeviceSupport) virtaddr=0xffffffff0085ef8b8 fileoffset=0x15eb8b8 size=0xee88
32 com.apple.iokit.IOSerialFamily (IOKit Serial Port Family) virtaddr=0xffffffff0085fe740 fileoffset=0x15fa740 size=0x6500
33 com.apple.driver.AppleOnboardSerial (AppleOnboardSerial) virtaddr=0xffffffff008604c40 fileoffset=0x1600c40 size=0x10500
34 com.apple.iokit.IOAccessoryManager (IOAccessoryManager) virtaddr=0xffffffff008615140 fileoffset=0x1611140 size=0x94060
35 com.apple.driver.AppleARMPMU (AppleARMPMU) virtaddr=0xffffffff0086a91a0 fileoffset=0x16a51a0 size=0x150d0
36 com.apple.driver.AppleEmbeddedTempSensor (AppleEmbeddedTempSensor) virtaddr=0xffffffff0086be270 fileoffset=0x16ba270 size=0x153a0
```

Figure 3: Sample Kextractor output

3.2 Kernelcache decryption

For versions up to iOS 10, the kernelcache is encrypted and must be decrypted before advancing to the next steps. This is done automatically if the header of the kernelcache contains the sequence **0x33676d49**. However, if the file needs to be decrypted, the user will be asked to input the Key and the IV necessary for the decryption algorithm. These can easily be obtained on numerous websites which are maintained by the iOS reverse engineering community. If the algorithm was successful, the decrypted kernelcache will be saved in the current directory.

3.3 Kernelcache decompression

As previously stated, kernelcache files come in a compressed form and must be decompressed before any extension can be extracted. Given that the compression algorithm has changed for more recent versions, Kextractor implements decompression algorithms for all iOS versions.

For versions up to iOS 14, Apple's proprietary algorithm LZSS was used. To begin the decompression process, the offset of the compression header before the segment containing kernel extensions must be determined. This header has the standard hexadecimal value of **0xFEEDFACE** or **0xFEEDFACF**. This offset is then used by the decompression script to apply the algorithm.

For more recent versions, the compression algorithm was changed to LZFSE. Similarly to LZSS, we must identify two offsets of the headers which indicate the beginning and the end of the compressed part. These headers have set hexadecimal values of **0x62767832** and **0x62767824**. The offsets are then passed on to the function that implements the decompression algorithm.

The implementation of the algorithms is the one publicly offered by Apple. If the algorithms were successfully applied, the resulting decompressed kernelcache will be saved at a location

specified by the user.

3.4 Usage demonstration

In order to successfully run Kextractor, you must install a list of dependencies specified in the official documentation[7]. After ensuring that all dependencies were obtained, you must run the install script. A usage guide is available to clarify the usage method.

```
usage: kextractor [-h] [-K KEXT] [-k] [-o OUTDIR] KCACHE

Kernel cache manipulation tool

positional arguments:
  KCACHE                path to decrypted kernel cache

optional arguments:
  -h, --help            show this help message and exit
  -K KEXT, --extract-kext KEXT
                        extract kernel extension from kernel cache
  -k, --show-kexts      show kernel extensions in kernel cache
  -o OUTDIR, --output-dir OUTDIR
                        store data to given directory
```

Figure 4: Usage guide

Running the Kextractor script requires the path of the kernelcache file as a parameter, resulting in a prompt asking for the path where the decompressed file should be saved.

```
roxana@main:~/Desktop/kextractor/kextractor/scripts$ ./kextractor -k kernelcache.release.iphone12c

Input path of output directory (make sure it ends with / ):
```

Figure 5: Usage guide

The successful run of Kextractor should provide the extracted extensions as an output. If the option to redirect this output to a specified file is selected, then a new file will be created.

```
roxana@main:~/Desktop/kextractor/kextractor/scripts$ ./kextractor -k kernelcache.release.iphone12c

Input path of output directory (make sure it ends with / ): /home/student/Desktop/

1 com.apple.iokit.IOSlowAdaptiveClockingFamily (IOSlowAdaptiveClockingFamily) virtaddr=0xffffffff008334988 fileoffset
2 com.company.driver.modulename () virtaddr=0xffffffff008336698 fileoffset=0x1332698 size=0x1a1f0
3 com.apple.iokit.IOReporting () virtaddr=0xffffffff008350888 fileoffset=0x134c888 size=0x2988
4 com.apple.driver.AppleARMPlatform (AppleARMPlatform) virtaddr=0xffffffff008353210 fileoffset=0x134f210 size=0x3

                ... (output truncated, 223 lines remaining) ...
```

Figure 6: Usage guide

4 Results and Limitations

Kextractor was initially limited to only extension retrieval, making it the user's duty to ensure that the kernelcache file was prepared for analysis. We then gradually added support for

decompression algorithms and integrated specific libraries and modules. Finally, we took the next step and added decryption features to complete the pre-analysis process.

Due to recent changes in kernel extensions and, consequently, in the kernelcache structure, Kextractor currently works for iOS versions up to iOS 16. Further analysis must be performed to determine a solution for this issue. Additionally, Kextractor is only available on Linux and macOS systems, which could limit the range of users. Lastly, because it has many external dependencies, bug fixing is more difficult and the rate of improvement is slower.

5 Conclusions

Kextractor is a tool meant to facilitate kernel analysis and the understanding of the power kernel extensions hold. The usage of these extensions, although it is extremely useful from a portability perspective, poses great security risks and must be more strictly managed. We consider Kextractor to be a stepping stone in that direction, allowing every user to be informed on what has access directly to the kernel.

5.1 Improvements and future work

Future improvements can be made to ensure support even for the latest iOS versions containing kernel alterations. Additionally, in its current presentation, Kextractor combines all features to offer a standalone tool for analysis. The separation of features can be achieved to enable the user to complete only intermediary steps in the kernelcache preparation process, such as only decryption or only decompression. Finally, an implementation that does not require any input from the user could be attained to provide an automated environment.

References

[1] J. Levin, “Mac OS X and iOS Internals, To the Apple’s Core”, John Wiley and Sons, Inc, 2013, pp 711-722.

[2] Apple Platform Security, “Kernel extensions in macOS and iOS”, Apple Inc, 2021, https://support.apple.com/guide/security/kernel-extensions-sec8e454101b/web.

[3] Apple Developer Support, “Code Signing”, Apple Inc, 2021, https://developer.apple.com/support/code-signing/.

[4] Charlie Miller, Dion Blazakis, Dino DaiZovi, Stefan Esser, Vincenzo Iozzo, and Ralf-Philip Weinmann. 2012. iOS Hacker’s Handbook. John Wiley and Sons.

[5] Răzvan Deaconescu, Luke Deshotels, Mihai Bucicoiu, William Enck, Lucas Davi, Ahmad-Reza Sadegh, SandBlaster: Reversing the Apple Sandbox, 2016, https://arxiv.org/abs/1608.04303

[6] J.Levin, jtool, http://www.newosxbook.com/tools/jtool.html

[7] Kextractor official documentation, https://github.com/malus-security/kextractor