BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INFORMATION SYSTEMS
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

CONTROL OF EXTERNAL DEVICES ON MACOS TO PREVENT DATA LEAKS

ŘÍZENÍ EXTERNÍCH ZAŘÍZENÍ NA MACOS S CÍLEM ZABRÁNIT ÚNIKU DAT

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

AUTHOR **Bc. JOZEF ZUZELKA**
AUTOR PRÁCE

SUPERVISOR **Ing. JAN PLUSKAL**
VEDOUCÍ PRÁCE

BRNO 2020

Brno University of Technology
Faculty of Information Technology

Department of Information Systems (DIFS) Academic year 2019/2020

Master's Thesis Specification

22637

Student: **Zuzelka Jozef, Bc.**
Programme: Information Technology Field of study: Information Technology Security
Title: **Control of External Devices on macOS to Prevent Data Leaks**
Category: Security

Assignment:

1. Analyze the basic architecture of the macOS operating system, especially in the context of handling external disk and network devices.
2. Discuss the current state of external devices control, extend the external device control topic by covering modern network drive such as cloud drives.
3. Propose different approaches to manage selected external devices after consulting with the supervisor.
4. Implement and demonstrate the most appropriate approaches.
5. Discuss the strengths and weaknesses of these approaches.

Recommended literature:

- Levin, J. (2013). Mac OS X and IOS Internals: To the Apple's Core.
- Wiley Halvorsen, O. H., & Douglas, C. (2011). OS X and IOS Kernel Programming: Master Kernel Programming for Efficiency and Performance. Apress
- Levin, J. (2019). MacOS and *OS Internals, Volume I: User Mode. Technologeeks Press
- Levin, J. (2019). MacOS and *OS Internals, Volume II - Kernel Mode. Technologeeks Press
- Levin, J. (2019). MacOS and *OS Internals, Volume III - Security & Insecurity. Technologeeks Press

Requirements for the semestral defence:

- Items 1, 2 and 3.

Detailed formal requirements can be found at https://www.fit.vut.cz/study/theses/

Supervisor: **Pluskal Jan, Ing.**
Head of Department: Kolář Dušan, doc. Dr. Ing.
Beginning of work: November 1, 2019
Submission deadline: May 20, 2020
Approval date: November 25, 2019

Abstract

This thesis is aimed at managing and blocking of external devices in `Apple macOS` operating system to prevent leaks of sensitive data. The implemented solution presents a chosen approach for blocking external drives and selected cloud drives. The project uses the `DiskArbitration` framework to block external devices, as it is the most suitable approach for this type of task. However, cloud drives are in reality just synchronized folders, therefore Endpoint Security framework had to be utilized to achieve an adequate level of control. Currently supported cloud providers are iCloud and Dropbox, and access to them can be restricted either entirely or to read-only. The ability to synchronize remote changes was preserved; however, in the case of Dropbox, its GUI cannot be used to edit files.

Abstrakt

Práca sa zaoberá problematikou kontroly a blokovania externých zariadení v operačnom systéme `Apple macOS` za účelom ochrany pred únikom citlivých dát. Implementované riešenie ukazuje zvolené prístupy pre blokovanie externých a cloudových diskov. Pre blokovanie USB diskov bol použitý `DiskAbitration` framework, čo je najvodnejšie riešenie tohto typu úlohy. Avšak cloudové disky sú v skutočnosti synchronizované zložky a úlohu nehrajú ovládače ani strom pripojených zariadení. Ku kontrole operácií v cloudových diskoch bol použitý `Endpoint Security` framework. Aktuálne podporovaní cloudový poskytovatelia sú iCloud a Dropbox a prístup k nim môže byť obmedzený úplne alebo iba na čítanie. Schopnosť synchronizácie vzdialenýh zmien bola zachovaná avšak v prípade Dropboxu si to žiada nepoužívať ich aplikáciu na správu súborov.

Keywords

Apple, OS X, macOS, Audit, Device Drivers, Kernel Extensions, System Extensions, IOKit, DriverKit, External Device, Device Control, Channel Control, DLP, Data Leaks Prevention, Kauth, MACF, KEXT, SYSX, iCloud, Dropbox, USB

Klíčová slova

Apple, OS X, macOS, Audit, Ovládače Zariadení, Rozšírenie Systému, Rozšírenie Jadra, IOKit, DriverKit, Externé Zariadenia, Kontrola zariadení, DLP, Prevencia Pred Únikmi Dát, Kauth, MACF, KEXT, SYSX, iCloud, Dropbox, USB

Reference

ZUZELKA, Jozef. *Control of External Devices on macOS to Prevent Data Leaks.* Brno, 2020. Master's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Jan Pluskal

Rozšířený abstrakt

Používanie externých zariadení je bežnou súčasťou každodennej práce s počítačom. Je to však tiež značné riziko kvôli možnému úniku citlivých dát spoločnosti. S rastúcim využívaním informačných technológií v bežnej obchodnej činnosti sa stáva viac a viac náročným ochrániť firemné tajomstvá a kontrolovať tok citlivých dát. V posledných rokoch sa stávajú bezpečnostné incidenty častejšie a množstvo spoločností či ľudí bolo zdiskreditovaných následkom úniku citlivých dát. Úniky dát spôsobené vonkajšími narušiteľmi sú však ďaleko menej časté ako úniky spôsobené radovými zamestnancami. Nemusí to nutne znamenať úmyselné vynášanie informácií. Napríklad zamestnanec, ktorý zverejní na verejnom úložisku súbor obsahujúci dáta, s ktorými pracuje na denno-dennej báze za účelom jeho zdieľania s kolegom, si nemusí uvedomiť následky svojich krokov až pokiaľ nie je neskoro.

Potreba riešenia týchto problémov vytvorila priestor pre riešenia zameriavajúce sa na úniky citlivých dát. Keďže väčšina informácií, s ktorými pracujeme je uložená v elektronickej podobe, pozornosť je zameraná hlavne na softvérové riešenia a metódy, ktoré detekujú potencionálne dátové úniky a predchádzajú im monitorovaním, detekciou a blokovaním citlivých dát.

Jeden z možných kanálov úniku sú externé úložiská. Pre predídenie úniku ich môžeme zablokovať úplne, na základe ich typov, či povoliť iba v režime pre čítanie. Cieľom tejto práce bolo navrhnúť a implementovať softwérové riešenie, ktoré by predchádzalo únikom citlivých dýt kontrolou týchto externých zariadení. Imlementácia úplného DLP riešenia vyžaduje vyriešiť radu prípadov, ako pripájanie zariadení počas štartu systému, využitie riešenia v reálnom čase (napríklad povolenie aktuálne zablokovaného zariadenia za behu systému bez nutnosti jeho reštartu), či podpora rôznych externých portov naprieč produktmi. Jednou z výziev počas výskumnej fázy práce bol nedostatok dokumentácie. Od poslednej verzie operačného systému sa odporúča použitie novo pridaného frameworku, ktorý nebol dostatočne zdokumentovaný a vyžadoval veľa skúšania metódou pokus-omyl. Práca sa viac zameriava na výskumnú časť a popisuje rôzne prístupy obmedzenia prístupu k systémovým prostriedkom. Implementačná časť sa bližšie venuje dvom typom externých zariadení — cloud diskom a USB diskom. Vedomosti získané v tejto práci by však mali postačovať na rozšírenie implementácie pre podporu ostatných poskytovateľov cloudového úložiska a typov externých zariadení.

Teoretická časť pozostáva z troch hlavných častí. Prvá časť popisuje históriu a architektúru operačného systému. Vysvetľuje ako vzniklo jadro a odkiaľ pochádzajú jeho jednotlivé súčasti. To pomôže nielen pri hľadaní informácií v zdrojoch k ostatným operačným systémom, ale tiež, ktorá komponenta je zodpovedná za danú funkcionalitu a čo od nej možno očakávať.

Druhá časť sa zaoberá popisom jednotlivých prístupoov a frameworkov, ktoré je možné využiť k sledovaniu a kontrole systémových operácií. Kedže kontrola vyžaduje viac informácií o stave a účele operácií ako čisté monitorovanie, text zahŕňa širší záber frameworkov, ktorých kombináciou je možné získať všetky potrebné informácie pre rozhodnutie o oprávnenosti danej operácie.

Posledná časť popisuje využitie vybraných frameworkov pre blokovanie konkrétnych zariadení. Pre blokovanie vymeniteľných diskov bol použitý `DiskArbitration` framework, ktorý poskytuje pomerne jednoduché rozhranie pre sledovanie a blokovanie vymeniteľných úložných médií. Tiež umožňuje pripojiť dané zariadenie so špecifikovanými parametrani, a teda napríklad v režime so zakázaným zápisom. Keďže cloudové disky nie sú systémom spracovávané ako pripojiteľné zariadenia, ale sú to zložky synchronizované so serverom klientskou službou, bolo treba zvoliť pre ich kontrolu iný prístup. K tomuto bol zvolený

`Endpoint Security` framework, ktorým sú súborové operácie kontrolované a zablokované v prípade, že nie sú oprávnené.

Podpora jednotlivých cloudov záleží na ich spôsobe akým synchronizujú dáta so serverovou časťou. Napríklad iCloud je synchronizovaný samostatným systémovým démonom, a teda je pomerne jednoduché rozlíšiť, či s dátami pracuje používateľ alebo sa synchronizujú so vzdialeným obsahom. Naopak Dropbox ponúka používateľovi možnosť správy súborov v rámci jeho aplikácie a na všetku prácu s nimi používa jeden proces. Ten istý proces tiež používa na synchronizáciu so serverom. Nebolo preto možné rozlíšiť, ktoré operácie patria k synchronizovaniu a ktoré vykonáva používateľ. Pre zaistenie správnej synchronizácie so serverom je teda nutné používateľom zakázať vstavaného správcu súborov Dropbox aplikácie. Kompletné riešenie bolo otestované na správnosť blokovania jednotlivých operácií a bol zmeraný časový dopad na vykonávanie jednotlivých operácií.

Control of External Devices on macOS to Prevent Data Leaks

Declaration

I hereby declare that this Master's thesis was prepared as an original work by the author under the supervision of Ing. Jan Pluskal. I have listed all the literary sources, publications and other sources, which were used during the preparation of this thesis.

....
Jozef Zuzelka
June 10, 2020

Acknowledgements

I would like to express my thanks to my supervisor Ing. Jan Pluskal for his patience, and dedicated time even despite his workload. I would also like to thank Ing. David Pořízek for his help during my work on this thesis. Finally, I would like to thank Ing. Zbyněk Sopuch for his guidance and the opportunity to work on this project.

Contents

Chapter 1

Introduction

External devices and their utilization for data transport is a standard part of daily work with a computer. However, while at first sight, it may appear as nothing extraordinary, external devices pose a serious threat of possible leakage of the company's sensitive data. With the increasing use of computer technology in standard business processes, it is becoming challenging to protect and control the flow of the company's sensitive data. In recent years, more and more security incidents occur, and many companies and people are discredited during these incidents by the data leaks. Data leaks caused by external intruders are far less common than leaks caused by the companies themselves and their employees. And it does not necessarily need to be caused intentionally. For example, when an employee uses a public service to share a file containing sensitive data with a colleague. The employee does not necessarily realize that he endangers company's security until it is too late.

The need to focus on these problems gave room to create solutions addressing data leak prevention. As most of the information is stored and manipulated in an electronic form, nowadays, the main focus is on software solutions and methods which detect potential data breaches and prevents them by monitoring, detecting, and blocking sensitive data.

One of the potential data leak channels is external storage. There are multiple approaches to how DLP products protect this channel. The protection is usually either restrictive or informative. The aim of this thesis is to propose and implement a software solution that would prevent leaks of sensitive data by controlling external devices. To implement a complete solution which could be later added to a DLP software, a couple of complicated use cases have to take into account, for example, connecting a device during OS boot, the usability of the solution in real-time (i.e., re-allowing a device during runtime without requiring a system reboot or other complicated procedures), or availability of different ports in different product models. One of the challenges during the research phase was the lack of official documentation from Apple. As of the last major update, the recommended approach leverages a newly added framework, which was not documented very well and required a lot of backtracking and trial-and-error testing. This thesis focuses on research more heavily and shows different ways of restricting access to system resources. The implementation part mainly covers two types of external storage — cloud drives and USB mass storage. However, the knowledge gained in this thesis should be sufficient for extending the implementation by other approaches to support also different types of cloud storage providers and device types.

Chapter 2 clarifies the naming conventions of the Apple operating system and describes their brief history and where core parts of the system came from. Individual parts are then described in more detail in chapter 3. `macOS` consists of many parts, and we can use different approaches to access external devices on different OS layers. However, not all approaches are suitable for a device control solution. Technical details and pros and cons of particular approaches are described in chapter 4, which describes different ways of leveraging kernel extensions for system monitoring and its control in more detail, chapter 5 describing system extensions which work from the user-space and are to replace kernel extensions, and finally, chapter 6 which covers monitoring of system events and communication with devices from user-space without the need of any extensions. Currently available projects working with external devices or the new frameworks are discussed in chapter 7, while usage of the various methods of blocking Dropbox, iCloud, and USB drives are covered in chapter 8. Chapter 9 then presents how implemented methods work in the real environment, and, finally, chapter 10 summarizes the results of the thesis.

Chapter 2

A Brief macOS History

In the source literature and other online sources, one can find Apple's operating system name in different forms. They are covered in this chapter and also their brief history. Apple's operating system presents an example of a successful connection of paradigms, ideas, and technologies that were not common to put together before. A nice example is a combination of a command-line interface with a graphical interface in macOS. The system is a result of many tries and tribulations of Apple, NeXT, and their user and developer communities. This chapter is mainly based on information from [65].

2.1 First Operating Systems in Apple Computers

The first Apple „computer" (shown in Figure 2.1) — `Apple I`, introduced in 1976, was just a motherboard, while a user had to get a case, display unit, ASCII-coded keyboard, and AC power sources on its own. It was without an operating system in today's perception of operating systems. There was just a 256 bytes-big firmware resident program (also called `Woz Monitor`[1]), which let users use the keyboard and the display to view memory content, to read and write programs, and so on. These programs were written as machine code[2] in hexadecimal format in MOS 6502's instruction set or programmed in Apple BASIC[3] provided on cassette with the computer[4] [3]. At the time, compared to UNIX, which already had the sixth edition, Apple's operating environment was significantly worse.

`Apple II` had the same environment with added support of more commands and color graphics. Shortly after `Apple II` release, Steve Wozniak designed a floppy disk drive and needed `Disk Operating System (DOS)`[6][7], which was released in 1978.

`Apple III` used a `Sophisticated Operating System (SOS)` as its system [64]. `SOS`[8] consisted of a kernel, an interpreter and set of drivers which were RAM-based and thus could be „installed", compared to ROM-based drivers in `Apple II`. `SOS` later evolved into `Apple ProDOS`.

[1] www.sbprojects.net/projects/apple1/wozmon.php
[2] www.willegal.net/appleii/apple1-software.htm
[3] www.atariarchives.org/mlb/chapter2.php
[4] www.sbprojects.net/projects/apple1/a1basic.php
[5] Image taken from http://www.breker.com/english/Apple_1.htm
[6] www.fabiensanglard.net/fd_proxy/prince_of_persia/Beneath%20Apple%20DOS.pdf
[7] Apple DOS was unrelated to Microsoft's MS-DOS
[8] pronounced „sauce", which makes it „Apple Sauce".

Figure 2.1: Apple I[5]

2.2 Pre-Mac OS X Systems

In 1983 and 1984, Apple released `Lisa` and `Macintosh` computers, respectively. These were highly inspired by work done at Xerox Palo Alto Research Centre (PARC), and most of the ideas used can be seen today — in macOS and other modern systems.

PARC was a home of new ideas and inventions like a drawing system for computer (Sketchpad — a photodiode pen used to interact with the computer), the first computer mouse, a five-finger equivalent of a full-sized keyboard, document processing, hypertext, search facilities, bootstrapping[9], high-quality graphical user interfaces (scroll bars, iconic and textual menus, overlapping and resizable windows, . . .), laser printing, windowing systems, networking (PARC Universal Packet – a predecessor of the TCP/IP protocol suite, Copy-Disk – protocol similar to FTP, . . .) and many more. For a more detailed description of all inventions and operating systems that influenced Apple's operating systems, a bonus chapter of a book written by Amit Singh — Mac OS X Internals: A Systems Approach [65] is an excellent source of information.

Lisa OS

The `Lisa` (Local Integrated Software Architecture) project began in 1979 with the release in 1983. The aim of this project was an intuitive, single-user, stand-alone, and easy to use microcomputer. As described above, `Lisa OS` (the Lisa Office System – a proprietary OS and a suite of office applications used in the Lisa computer) was highly inspired by work done at Xerox PARC. Apple claimed the user interface was such intuitive a first-time user could do productive work within 30 minutes while existing computers required 20-30 hours of training and practice. Although `Lisa` was a commercial failure, technologically, it introduced several aspects that were used in latter Apple systems [64].

„Classic“ Mac OS

The successor of the `Lisa`, the `Macintosh`, was unveiled in 1984. It was cheaper and better marketed than `Lisa`. The `Macintosh` ran a single-user, single-tasking OS, initially known

[9] `www.dougengelbart.org/content/view/226/269/`

as Mac System Software. Unlike Lisa, the Macintosh was not designed to run multiple OS'. The Macintosh ROM contained low-level code for hardware initialization, diagnostics, drivers, and so on. The higher-level part was a collection of software routines meant for use by applications, like a shared library.

The operating system of the Macintosh is also referenced as System 1. After Macintosh's release, Apple spent the next few years improving the Macintosh operating system and creating other systems, including System 2-6, A/UX, Apple Workgroup Server, and others. Detailed information about these systems can be found in [65].

In 1997, a major upgrade over System 6 was released, named System 7. Mac OS 7.6 dropped the „System" moniker, and the last two major releases of Mac OS before Mac OS X were Mac OS 8 (initially planned as Mac OS 7.7) and Mac OS 9. Mac OS X supported Mac OS 9 and its applications up to version Mac OS X 10.4 Tiger.

NEXTSTEP

After Steve Jobs was forced to leave Apple in 1985, he founded a startup NeXT Computer, Inc. with other five Apple employees. The computer they developed ran an OS called NEXTSTEP, which was unveiled in 1988. The NEXTSTEP used a port of Carnegie Mellon University (CMU) Mach 2.0 with a 4.3BSD environment. NEXTSTEP used Objective-C as the main language of the platform.

NEXTSTEP (later, since version 4.0, known as OPENSTEP) offered several software kits, which were collections of reusable classes for different areas of development (i.e., the Application Kit, the Music Kit, and the Sound Kit). In later versions of the operating system, application development became easier thanks to an extensive collection of libraries for user interfaces, databases, distributed objects, multimedia, networking, and so on. NEXTSTEP also had an object-oriented device driver toolkit for driver development.

After Apple bought NeXT in 1997, it announced it would base its next operating system on OPENSTEP. It inherited NeXT's technologies like software kits, Mach kernel, and Objective-C became an essential language for Apple as well.

2.3 Mac OS X, OS X, and macOS Era

After acquiring NeXT, Apple based its next-generation operating system on Next's OPENSTEP. This was developed into Rhapsody in 1997, Mac OS X Server 1.0 in 1999, Mac OS X Public Beta in 2000, and Mac OS X in 2001. The X in its name is a roman numeral corresponding to the tenth version of Mac OS. Apple also introduced Darwin – a fork of Rhapsody's developer release, which would become a core of Apple's systems. Darwin is described in more detail in chapter 3 [64].

Mac OS X is still the latest major release of Mac OS, however, Mac OS X was officially renamed to OS X in 2012 with the release of OS X 10.8 „Mountain Lion" as a link to a refinement of the previous Mac OS X version. Another renaming occurred in 2016 when Apple released 10.12 Sierra, and OS X was changed to macOS, probably to follow the naming convention of its other systems (i.e., watchOS, iOS, tvOS, iPadOS, etc.). A photo of the latest version can be seen in Figure 2.2.

Over the time, there were several system protections added (i.e., SIP, Kernel Extensions (KEXTs), System Extensions (SYSXs), Sandboxing, Code signatures, moving the system to read-only partition, etc.), which also affected implementation a device control.

[10] Image taken from www.apple.com/newsroom/2019/06/apple-previews-macos-catalina/

Figure 2.2: macOS Catalina[10]

`OS X 10.11 El Capitan` introduced a new security feature called `System Integrity Protection (SIP)`[11]. This feature added several mechanisms enforced by the kernel to protect system-owned files and directories against modification by unauthorized processes, even when executed by the `root` user. Protection is enabled by default and can be disabled, although it is not recommended. This feature can only be disabled from outside of the system partition, i.e., from the recovery system or bootable `macOS` installation disk [71].

Another security feature that influenced device drivers' development is System Extensions and other system restrictions[12] introduced in `macOS 10.15 Catalina`. Until then, one could use a Kernel Extension to extend the functionality of the kernel. Since `OS X Yosemite`, kernel extensions, such as drivers, have to be code-signed with a particular Apple entitlement, and currently are deprecated and their support will probably be removed in the next `macOS` release.

However, Kernel Extensions still can also be used in `Catalina`. First, it has to be signed and then notarized[13], which means it has been checked by Apple for malware. Then it has to be manually added into „User-Approved Kernel Extensions" in system settings, but it is still not all. `Catalina` also introduced the read-only partition of the operating system, which means that the kernel extension build into the pre-linked kernel needs to be stored on the read-only System Volume. There are two occasions when the System Volume is writable – during a `macOS` update and just before `macOS` shuts down. Thus, OS needs to be restarted to load the new pre-linked kernel [59], which limits the usability of kernel modules. Kernel extensions are discussed in more detail in chapter 4 and system extensions in chapter 5.

[11] SIP is sometimes called also `rootless`.
[12] www.sentinelone.com/blog/7-big-security-surprises-coming-to-macos-10-15-catalina/
[13] www.developer.apple.com/documentation/xcode/notarizing_macos_software_before_distribution

Chapter 3

Architecture of macOS

A proper understanding of how the operating system works is undeniably helpful in designing, developing, and debugging programs by developers of various experiences. This chapter covers the architecture of `Apple's macOS` operating system and describes its fundamental parts. The `macOS` consists of several parts that can be seen in Figure 3.1. Higher the layer is — less specialized the technology in the layer is. Generally, these layers use lower layer technologies to offer its functionality.

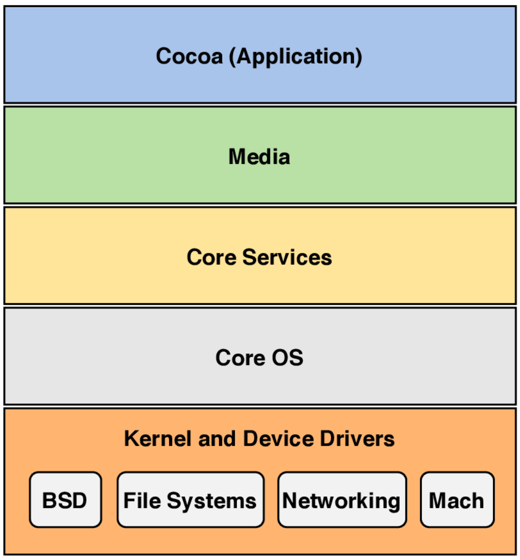

Figure 3.1: Layers of macOS [20]

The Cocoa layer includes technologies used for an app's user interface development, including, for example, Notification Center notifying a user for various application related events. The Media layer contains technologies for working with audiovisual media and graphics, supporting more than 100 media types. The Core Services layer consists of many fundamental services and technologies, such as Automatic Reference Counting, string manipulation, and data formatting. This layer also contains some higher-level features like iCloud integration, Grand Central Dispatch, Security Services, XML and SQLite technologies, Distributed Notifications, and many more [20].

The Core OS layer is responsible for programming interfaces related to hardware and networking based on facilities in the Kernel and Device Drivers layer. The layer takes care of app security-related technologies like Gatekeeper, App Sandbox, and Code Signing, but it also contains one feature especially useful for device control implementation called Disk Arbitration, which can be used to audit but also to manage external devices. This framework is described in greater detail in chapter 6.

The lowest layer, the Kernel and Device Drivers layer consists of the `Mach` kernel environment, device drivers, `BSD` library functions, and other low-level components [20]. All of these are part of a minimal operating system called `Darwin` described further.

Darwin

After Apple bought `NeXT` and released `Mac OS X Public Beta` in 2000, it released its core components as open-source software, `Darwin`; however, the higher-level components, such as the Cocoa and Carbon frameworks, remained closed-source.

`Darwin` is an operating system by itself and is also used as the core OS of other Apple's operating systems, such as `iOS`. It is a collection of technologies that Apple integrated together to form a central part of `macOS` containing many packages from Apple, but also many others such as `BSD`, or `GNU`. `Darwin` includes a kernel and a set of userland applications, but it does not include higher-level frameworks like Cocoa and Carbon. So it does not include `macOS`' windowing system `Aqua`. On top of `Darwin` are various proprietary systems that combine to form `macOS`.

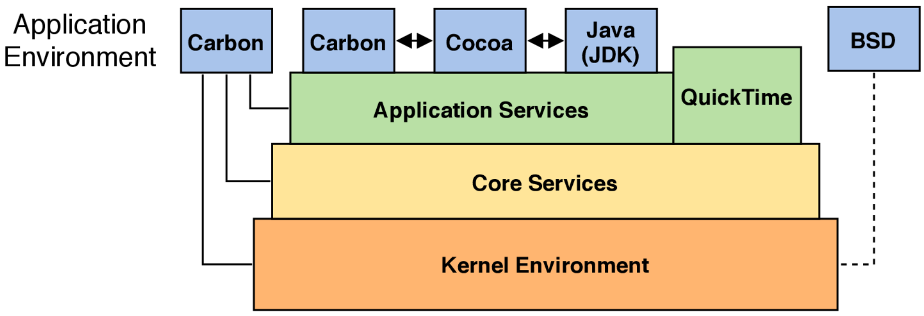

Figure 3.2: Layers of macOS [17]

The most noticeably unique userland feature of `Darwin` is `launchd`. It is the first process that is started by the kernel and can run tasks in response to network activity, timers, and so on. The rest of the userland is a hybrid of `FreeBSD`, `OPENSTEP`, and `GNU` utilities (e.g., Apache, Samba, GNU Make, clang, GNU bash, etc.) [34]. Among other features `Darwin` supports are SMB network file system, multicast DNS responder, or Bonjour networking technology.

Figure 3.2 illustrates general `macOS` architecture, while `Darwin` parts are highlighted in Figure 3.3 to show the relationship between `Darwin` and `macOS`. Both of them contain the `BSD` command-line application environment, however, in `macOS`, the environment is hidden and the user does not have to use it unless they choose to. The kernel of Darwin is `XNU`[1] and is described in section 3.2. `macOS` internals can be grouped into three logical layers: the firmware, the kernel, and the rest above the kernel.

[1] `XNU` is unofficially an acronym for „X is Not Unix"

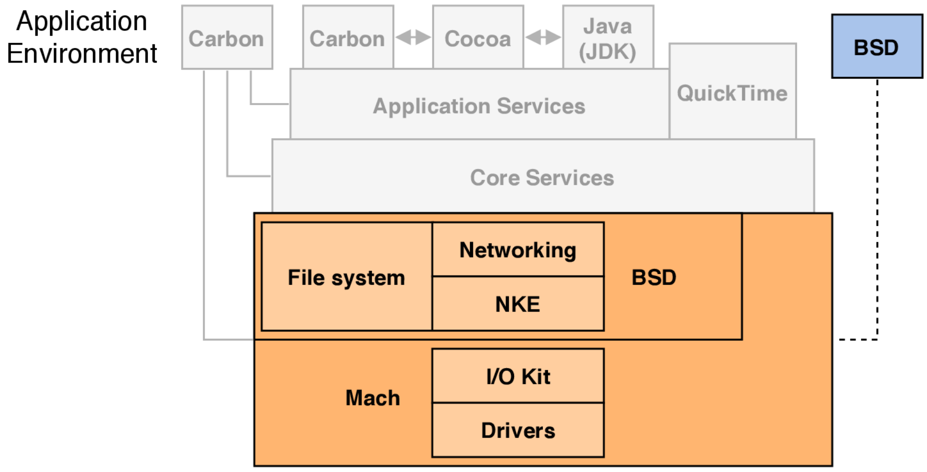

Figure 3.3: Layers of Darwin [17]

3.1 Firmware

The firmware and the boot loader are not technically parts of `macOS`, but it is briefly covered in this section as it plays an essential role in the operation of the machine and is useful in debugging. This section is based mainly on [49]. After the machine is powered on, at the most nascent stage — before the CPU starts executing the operating system code — the CPU executes standard startup code which probes hardware, finds the operating system, and prepares the machine for booting the system.

Intel-based machines traditionally relied on a (very) Basic Input Output System — a `BIOS`, whereas `PowerPC`, like many other systems used firmware. The `BIOS` (or rather `BIOS User Interface`) provided a set of simple menus by means of which the user could change motherboard parameters, boot device order, and other settings. From the other end, a `BIOS Processor Interface` provided an interface for CPU to invoke its functions, i.e., for device I/O [49].

At the very beginning, both firmware and `BIOS` load the CPU with a basic bootstrap code responsible for the `POST` (Power-On-Self-Test) phase. In this step, the CPU initializes various hardware interfaces and verifies that sufficient memory is available. Then it localizes the boot device, and executes a boot loader program, which in turn finds the operating system, and passes its kernel needed command-line arguments.

Before Apple's Intel days, their machines used `PowerPC` architecture and employed a custom firmware called `OpenFirmware` and `BootX` as the boot loader. Various limitations of `BIOS`, and others, like the support of only a simple disk partitioning scheme, limit of its maximal supported size, impossibility to interface with today's graphics, or only one possible active system led Apple to adopt a newer compatible standard of the Extensible Firmware Interface — `EFI`. In 2005, Apple announced its transition to Intel processors[2], and dropped `PowerPC` support starting `Mac OS X 10.6 Snow Leopard` and as the first major OS adopted `EFI`. `EFI` is a runtime environment that offers a more capable interface during boot and even later during runtime. `XNU`, the `macOS` kernel, relies on many of `EFI`'s features, one of which is, for example, `NVRAM` variables. Apple slightly changed its `EFI` implementation. For example, it is wrapped with a custom header to use the same binary for 32-bit and

[2] www.apple.com/newsroom/2005/06/06Apple-to-Use-Intel-Microprocessors-Beginning-in-2006/

64-bit architectures. Additionally, most EFI implementations provide a shell interface, but Apple's implementation only responds to specific key presses that the user should input after the system startup sound [49].

EFI started as an initiative by Intel and later merged with an open standard called Universal EFI -- UEFI. Apple's implementation is compliant with EFI 1.10, which was its final version, but also implements some features from UEFI. Not all Apple products use the same firmware, however. Although UEFI is processor-agnostic and has implementations for both Intel and ARM, iOS uses a custom boot-loader, iBoot, which is not EFI-based. More details about macOS' EFI architecture, its services, and its flow of initialization can be found in great detail in the book from Jonathan Levin: Mac OS X and iOS internals: To the Apple's Core [49].

Both POST and EFI are provided by BootROM, which is a part of computer's hardware. Once BootROM is finished, control is passed to the boot.efi boot loader. The boot loader takes care of gaining a password from user in case of encrypted disks, showing boot image, and loading the operating system itself [17]. One of the steps in the boot process is InitDeviceTree. In this step, a hierarchical, tree-based representation of the devices — called Device Tree — is created and later passed to the kernel. The kernel itself does not work with this structure much, but the I/O Kit subsystem relies heavily on it. The device tree is visible in I/O Kit through a special plane called the IODeviceTree plane. This plane can be shown using command `ioreg -w 0 -l -p IODeviceTree | grep -v \"IO` , or using Apple's I/O Registry Explorer tool.

I/O Kit, together with the concept of planes, is described in greater detail in section 3.2.3, and the process of creation of the device tree is discussed in more detail in chapter 4. In the next steps, EFI allocates memory for the kernel where it is loaded from the boot-device, checks if the system needs to be resumed from hibernation, processes boot options, which are later passed to the kernel command line, and locates and loads a prelinked kernel. Then inits a boot structure containing all the parameters kernel needs (from its command-line arguments to the device tree) and loads various device drivers — KEXTs — into the kernel. Finally, after several other steps, the control is transferred to the kernel, passing it a single argument — the Boot-Struct[3] [49]. Once the kernel and all drivers necessary for booting are loaded, the boot loader starts the kernel's initialization procedure which inits Mach and BSD structures (both subsystems are described in the following sections). Afterwards, I/O Kit uses the mentioned Device Tree to determine which loaded drivers to link into the kernel [17]. In the end, the top-level file system is mounted and startup scripts executed[4]. Detailed visualization of the boot process can be found in [58].

3.2 XNU: The Kernel

macOS' kernel environment is mainly built on top of two parts, Mach 3.0 and FreeBSD, but also contains code and concepts from other *BSD derivatives and other Apple projects, such as the MkLinux project. Figure 3.3 shows architectural components of the kernel environment.

BSD provides basic file system and networking services, and a user and group identification schemes. BSD also enforces access restrictions to files and system resources using user and group IDs. Mach is the foundation of the OS providing memory management,

[3]The Boot-Struct structure can be found in the kernel sources (pexpert/pexpert/i386/boot.h)

[4]In some literature, the last step of starting the OS — mounting the root partition — is called rooting and is not considered as a step of booting.

thread control, hardware abstraction, and Inter-Process Communication (IPC). Mach ports represent tasks and other resources, and Mach enforces access to those ports by checking if tasks are permitted to send a message to them [21].

Originally, Mach was a hybrid kernel developed by NeXT for NEXTSTEP based on Mach 2.5, 4.3BSD, and an Objective-C API for driver development — Driver Kit. After Apple bought NeXT and used XNU for Mac OS X, Mach 2.5 was upgraded to OSFMK7.3, 4.3BSD to FreeBSD, and Driver Kit was replaced by a C++ API I/O Kit. The following sections describe individual kernel parts in more detail.

3.2.1 Mach Kernel

As mentioned in previous sections, Mach kernel was used in NEXTSTEP and it also made its way to Mac OS X. The Mach project started in 1984 as a successor of not so successful Accent kernel and Rochester's Intelligent Gateway (RIG) system. Some of the goals during the kernel development were to provide full support for multiprocessing, reduce the number of features in the kernel to make it less complex, provide compatibility with UNIX, and address shortcomings of previous systems. When Mach was developed, UNIX had been out for over fifteen years [65].

The Mach project was partially a response to the increasing complexity of UNIX, which was no longer as simple or easy to modify as it used to be. Mach's implementation used 4.3BSD as the starting codebase and as it evolved, portions of the BSD kernel were replaced by their Mach equivalents, and various new components were added [65]. Mach provides the basic abstractions for getting the system running, such as processor and memory abstractions. Mach intends to be policy-neutral and leaves the policy decisions to higher levels of the software. In the case of macOS, a lot of these decisions are implemented in the BSD layer and some even higher, i.e., in the Classic layer. Mach is just a foundation for building operating systems on top of it, not an OS. It was not meant to provide neither provides any I/O capabilities (which are provided by I/O Kit described in section 3.2.3), Networking, nor File system services (which are implemented in the BSD layer covered in section 3.2.2). It does not do any security policies either. Although it provides lots of security mechanisms, it makes no decisions about how they get applied — many policy decisions are implemented in the BSD layer [7].

A lot of research came with a lot of Mach kernel versions. Some of them were monolithic, such as Mach 2.x with hardcoded pieces of BSD in it (i.e., for I/O operations), and others with a more modular architecture, like Mach 3.0 where the hardcoded dependencies from Mach 2.5 became formalized interfaces. Mac OS X (Darwin 1.x) was based on Mach 3.0 by OSF (OSFMK 7.3)[5] [7]. The Mach component is responsible for most of the lower-level functionality, such as Virtual Memory Management (VMM), IPC, preemptive multitasking, protected memory, and console I/O. Also inherent in the design of XNU are the Mach concepts of tasks, rather than processes, containing several threads, and the IPC concepts of messages and ports [61].

Each macOS application is a BSD process with two basic sets of resources — a Mach task and File Descriptors as is shown in Figure 3.4. A task is a unit of Mach resource ownership. A task is also an environment for threads (primary units of execution that are then scheduled by the OS), providing them VM address space and port namespace. A Bootstrap server is used to distinguish different tasks between each other (i.e., a Java task and a Carbon task). In Mach, a thread is just a register state and scheduling attributes.

[5] Mac OS X Server 1.x (Darwin 0.3) was based on Mach 2.5 with added features from Mach 3.0.

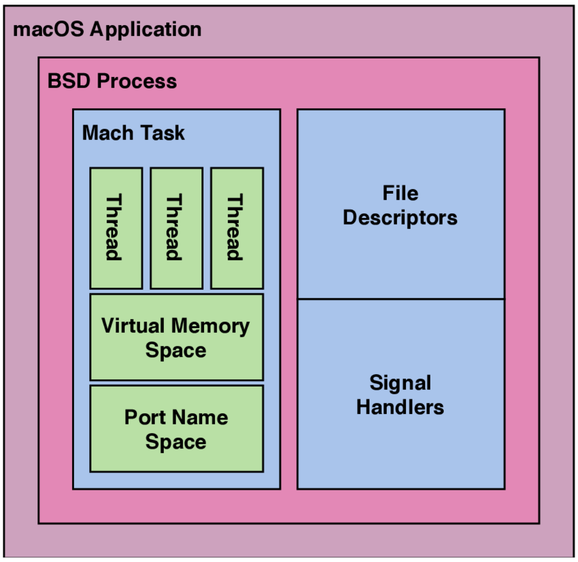

Figure 3.4: The architecture of a BSD Process and a Mach Task [7]

Thus there is no stack. These layers are implemented in other parts of the system as is shown in Figure 3.5. The various threads of a task share its resources, although each has its own execution state including the program counter and various other registers. Every `POSIX` thread has a corresponding `Mach` thread. Typically, this statement is also valid in the opposite direction, but technically, it is possible to create a `Mach` thread without the `POSIX` thread [7].

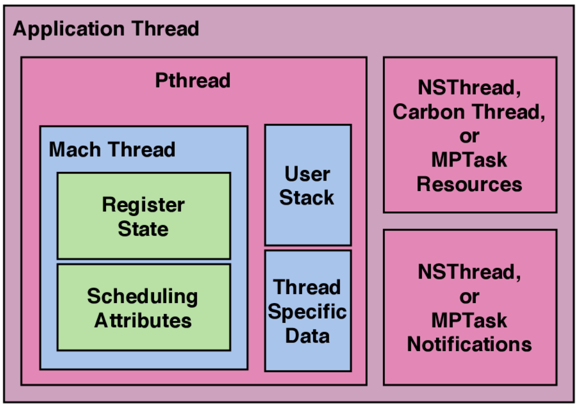

Figure 3.5: The architecture of a Pthread and a Mach Thread [7]

Unlike most other `Mach`-based operating systems, such as `BSD HURD`, `XNU` implements UNIX system calls in the same way as a `BSD` system. This is primarily for performance as it is much faster to make a direct call between the kernel components than to send messages from user-space[6] [34]. Moreover, the primary advantage of running `Mach` separately and not affecting it by, for example, a crash of the `BSD` part is not really an advantage, as the `BSD`

[6]Traditional `UNIX` systems had a single process for the kernel, which meant that everything the kernel was responsible for was part of a single binary, with no protection between the various parts. The goal of the `Mach` kernel was to separate out all the parts and provide a mechanism for joining them together. This brought significant slowdown because of running tasks in parallel and their communication using message

layer is critical for the majority of running applications as well as other components of the kernel. Thus in macOS, Mach is not primarily a communication hub between clients and servers but is used as an abstraction layer and is linked with BSD and I/O Kit into a single kernel address space [6]. However, Mach messages are used for many things in macOS – e.g., delivery of events from the user. Other processes can use Mach-level IPC as well. This brings some advantages like the possibility to check the other party in Mach communication, which comes handy when implementing applications like the Keychain [34].

3.2.2 BSD Kernel

Atop Mach sits a modified BSD kernel now based on FreeBSD, and code flows both to and from this project on a fairly regular basis [34]. However, they are not the same. For example, as the Mach layer is responsible for threads, it is also responsible for scheduling, so FreeBSD scheduler is not present in macOS. The BSD component is responsible for implementing a POSIX-compliant API, multi-user access, TCP/IP networking, memory protections, implementation of the UNIX process model (pid/gids/pthreads) on top of the equivalent Mach concepts (task/thread), Virtual File System (VFS), and so on.

BSD extends Mach kernel and provides process model, basic security policy model, file system and networking architecture but also userland libraries and services available to the applications. At the very lowest layer, Mach is responsible for the abstraction of the processor and the memory management as described in section 3.2.1. The BSD process sits on top of those very low-level primitives and provides additional things like OS resource management and other ancillary services, such as interrupt dispatch. It is responsible for a process' file descriptors, provides high-level memory abstraction as well as all the network resources. When a BSD process owning these resources terminates, the BSD component is responsible for reclaim all the resources associated with the process [8].

BSD security policy model brings the concept of users, each with their own set of privileges and capabilities. Another aspect of security policy involves the file system where each file on the system has an owner and a group that is part of permission access to those files associated with it. BSD as also an environment within which the file system sits. It is based on a VFS architecture supporting a number of different file system types [8].

3.2.3 Others

There are more parts in the kernel that are neither directly in the Mach component nor in the BSD component, for example, I/O Kit and Kernel Extensions. These parts are more described in the rest of the section.

I/O Kit

I/O Kit is a complete, self-contained execution environment in the kernel, but it is also a framework. In macOS, device drivers are written in a restricted subset of C++, which omits many C++ features that cause runtime overhead, or that might cause problems in the kernel space, such as exception handling, multiple inheritance, and templating. Compared to OPENSTEP, which was a base for Mac OS X as described before, Apple decided not to use Driver Kit framework which used Objective-C and chose to use a dialect of C++. I/O Kit, the framework used for writing device drivers, provides a hierarchy of C++ classes

passing (which was the only form of IPC in Mach). Every Mach message send required checking the sending and receiving port access rights and complex memory mapping operations.

for various generic drivers and allows drivers to be written by subclassing them. I/O Kit also implements a registry system in which all instantiated objects are tracked, as well as a catalog database of all the I/O Kit classes available. The development of a driver and parts of I/O Kit is discussed in a separate section of chapter 4. The latest release of macOS – Catalina – introduced a new environment for device drivers development called DriverKit[7] described in more detail in chapter 5.

Kernel Extensions

XNU is a modular kernel so it supports loadable kernel modules. To store them at a disk, Apple developed kext as a file format. Rather than a single file, kext is a directory containing several files, including a loadable object file (in Mach-O format) extending the functionality of the macOS kernel.

Usually, there is no need for developing kernel extensions as the functionality available in the user-space is sufficient for most tasks. However, there are tasks that cannot be implemented without a kernel extension. For example, the ability to dynamically add a new file system implementation is based on VFS KEXTs. Device drivers and device families in the I/O Kit are implemented using KEXTs as well. In the latest release of macOS — Catalina — Kernel Extensions became deprecated and should be replaced with System Extensions. Both are discussed in greater detail in the following chapters.

File Systems

macOS provides support for numerous types of file systems, including HFS, HFS+, APFS, NFS, and others. The default file system was changed from HFS+ to APFS in macOS High Sierra. The file system component also includes advanced features, such as enhanced VFS design [6]; however, this topic will not be covered in this thesis. File system is a subsystem of the BSD environment, which was described in section 3.2.2.

Networking

macOS networking takes advantage of BSD's advanced networking capabilities to provide features, such as Network Address Translation and firewalls. The networking component provides support for TCP/IP stack and socket APIs, IP, AppleTalk, multi-homing, routing, multicast support, and more [6]. Networking is a subset of the BSD system as well, built on top of BSD socket APIs.

3.3 Layers Above the Kernel

This section discusses software layers above the kernel, which can be seen in Figure 3.2. Core Services is a set of macOS and iOS APIs that architecturally are below Carbon, Cocoa, and Cocoa Touch. The most essential components of this layer are CoreFoundation.framework and CoreServices.framework which contain critical non-GUI system services (i.e., managing threads and processes, virtual memory, file system interaction, networking, . . .) [64].

The Application Services layer provides services for graphics and windowing environment of macOS. In this layer, we can find, for example, Quartz, which is the core of the windowing environment and a part of CoreGraphics.framework. Furthermore, it includes 2D

[7] DriverKit frameworks in Catalina and OPENSTEP are two different things, although both are/were used for drivers development.

renderer, the composition engine communicating with the graphics card, and the hardware acceleration layer. The Application Services layer also includes various other components, such as a printing subsystem, mechanism for inter-application communication, a framework for accessing and managing fonts, and others. In higher layer is Application Environments containing multiple execution environments, such as `BSD`, `Carbon`, or `Cocoa`. More details about these particular environments can be found in [64].

3.4 Summary

Previous sections briefly described core parts of the `macOS` architecture. The `macOS` system can be divided into several logical layers based on their functionality, such as the kernel, Core OS, media layer, or the Cocoa layer. Technically we can split the system into the firmware, the kernel, and the rest. The base of the *`OS` systems from Apple is `Darwin` — an operating system within the operating system that consists of several user-space services and the `XNU` kernel. The kernel has three main parts — the `Mach` kernel, the `BSD` kernel, and the `I/O Kit`. All three parts reside together in the kernel-space and each is responsible for some part of the `XNU`'s functionality.

From a DLP developer's point of view, the most important information in this section is the overall architecture of the system and the relation of the subsystems such as the representation of the same process in both `Mach` and `BSD` components. This section also lightly discussed kernel extensions, driver development environment, and instantiation and initialization of device drivers starting from the boot. Because both kernel extensions and the `I/O Kit` are crucial for work with external devices (especially for the altering of communication with them), they are described in more detail in chapter 4.

Chapter 4

Accessing System Using Kernel Extensions

macOS provides a kernel extension mechanism to allow dynamic loading of pieces of code into the kernel, without the need to recompile it. These pieces of code are known generically as `plug-ins` or in the macOS kernel as `Kernel Extensions` — KEXTs [6]. KEXTs perform low-level tasks that cannot be performed in user-space. Typically, a KEXT can be placed into one of three categories: Low-level device drivers, Network filters, or File systems. This section is mainly based on information from [11, 17].

Using KEXTs, one can extend the built-in functionality of the operating system. However, KEXTs can be challenging to develop and debug, and they can be a risk to data security and privacy. Not well-tested KEXTs can also cause system instability and crashes, so it is crucial to test them thoroughly. As KEXTs run in the same address space as the kernel, a single bit written into the wrong memory address can bring down whole operating system. Also, if a KEXT is loaded at system startup and contains an error, it will crash the system each time it starts, further complicating system recovery [17].

Among typical reasons for writing kernel-resident code instead of user-level application or plug-in are if the code provides a resource that is being required by a large number of applications, if code's primary client resides in the kernel (i.e., file system and networking device drivers), or if the code needs to handle primary interrupts (i.e., network controllers, graphics drivers, audio drivers, etc.). Though, not all drivers require a kernel extension. For example, for developing a USB or FireWire device driver, IOKit provides an interface for communication with the devices from user-space [11].

During the development of kernel extensions, one has to keep in mind several aspects:

- Kernel extensions reduce the amount of memory available to user programs, as kernel-space cannot be paged out, thus requires wired memory.
- The kernel runtime is more restricted environment than user-space, and code needs to follow those restrictions to avoid errors.
- Bugs in a code in the kernel-space are far more severe and can cause system instability and crashes.
- Debugging kernel code is more difficult, as stopping the kernel causes to stop the debugger itself as well. Thus it requires two-machine setup with a remote debugger.
- Some customers may prohibit using of 3rd-party KEXTs.

KEXTs are loadable bundles, which means they are loaded dynamically by another application but also has to follow the structure of a bundle. They have strict security and location requirements which have to be followed. Every KEXT has to contain Information Property List, Info.plist, containing information about it, such as bundle identifier, list of other KEXT libraries it links against, I/O Kit personalities for automatic loading of drivers, and others. The plist file has to be in XML format without comments as the KEXT can be loaded during early boot when only limited processing is available. KEXT usually also contains an executable file and optionally its resources and plug-ins. The executable is responsible for defining entry points that allow the KEXT to be loaded and unloaded. These entry points differ based on executable's type, which can be of two types — a generic kernel extension or an I/O Kit driver. The kernel does not differentiate between KEXTs containing either of the types, and it can incorporate elements of both (though one has to be careful, as each of them uses different threading models, memory management models, locking models, and so on). Both types of kernel extensions are discussed later in this section.

In order to load and install a kernel extension with SIP enabled, the KEXT has to be signed with a developer ID certificate dedicated for kernel extension and installed into /Library/Extensions directory. Moreover, KEXT bundle has to have proper permissions set for all files and folders. The owner has to be the root user with the wheel group, and the root being the only one granted write permissions.

Generic Kernel Extensions

Generic Kernel Extensions are usually written in C programming language. The extension has to be loaded and unloaded explicitly, either using kextload command or by a system reboot. Entry points of the extension are start and stop functions with C linkage, and its functionality is implemented using registered C callbacks with relevant subsystems. Once the KEXT is loaded it can use Kernel Programming Interfaces (KPI), such as Kauth, and MACF.

Listing D.1 shows a simple generic extension. The most important parts are its endpoints — start and stop functions. Usually, kernel extensions register and unregister callbacks with kernel runtime systems, as can be seen in later sections of this chapter. However, this example just prints a debug message to confirm it has been loaded. Working Xcode project using a generic KEXT can be found in the thesis' source files.

I/O Kit Kernel Extensions

I/O Kit Kernel Extensions use a subset of C++ language. They are loaded and unloaded automatically by the I/O Kit when needed; however, there is currently no easy way of installing an I/O KEXT without a need of reboot, and even Apple's own installers require restart of the system [13]. As extension entry points are used static C++ constructors and destructors. Functionality is implemented using subclasses of I/O Kit driver classes, such as IOUSBDevice. The I/O Kit is described in greater detail in section 4.4.

Codeless Kernel Extensions

Codeless Kernel Extension is a type of KEXT that does not contain an executable, typically used to tell I/O Kit what (existing) driver to use for the particular device. Listing 4.1 contains an example of a codeless KEXT's IOKitPersonalities dictionary which names other KEXTs that are loaded when a personality matches on a device. In this case, it's

Apple's generic driver `com.apple.driver.AppleUSBMergeNub`. Codeless `KEXT`s are usually used with USB and HID devices that are not included in Apple's generic driver's matching dictionary but work well with it.

```
<key>IOKitPersonalities</key>
<dict>
    <key>My_USB_Printer</key>
    <dict>
        <key>CFBundleIdentifier</key>
        <string>com.apple.driver.AppleUSBMergeNub</string>
        <key>IOClass</key>
        <string>AppleUSBMergeNub</string>
        <key>IOProviderClass</key>
        <string>IOUSBInterface</string>
        <key>idProduct</key>
        <integer>0000</integer>
        <key>idVendor</key>
        <integer>0000</integer>
    </dict>
</dict>
```

Listing 4.1: Codeless KEXT Personalities [11]

4.1 IP/Socket Filters

`XNU` provides a possibility of receiving of network and IPC events using kernel's socket layer. Socket filters are part of `Network Kernel Extensions (NKE)`. The framework provides a way of registering their callbacks (for IPv4 and IPv6) of receiving a subset of events they are interested in (i.e., TCP or UDP flows). Particular callbacks that can be registered can be found in `bsd/sys/kpi_socketfilter.h`. Among others, there are callbacks to be notified about opening and closing sockets [80]. An example implementation of network socket filter can be found in Apple's archive[1].

The second framework is IP filter which one can find in `bsd/netinet/kpi_ipfilter.h`. This framework provides a way of filtering incoming and outgoing packets. This thesis does not further cover area of network events, but more information can be found in [15] and [55]. It's important to note that `Network Kernel Extensions` were replaced by `Network Extensions` as a part of the transition to `System Extensions` in `macOS Catalina`, and are briefly describe them in chapter 5.

4.2 Kernel Authorization

Together with `Mac OS X 10.4 Tiger`, a `Kernel Authorization` framework (`Kauth`) was introduced. It was a new kernel subsystem dedicated for management of authorizations within the kernel of the system. Although this subsystem itself does not improve system security, it provides `Kernel Programming Interface (KPI)` providing developers a way to authorize operations within the kernel and extend its area of authority. It can also be used just to passively monitor system events. `Kauth` was later also implemented into

[1] https://developer.apple.com/library/archive/samplecode/tcplognke/Introduction/Intro.html

NetBSD 4.0[2]; however, due to licensing issues, developed from scratch based on Apple's documentation [38]. A certain similarity can be seen with Linux Security Modules (LSB) and TrustedBSD/FreeBSD Mandatory Access Control (MAC) framework. This section is mainly based on information from [12].

Although Kauth was initially designed to simplify the implementation of Access Control Lists (ACL), it is a general authorization mechanism of the kernel that can be used for various purposes. Kauth introduces several fundamental concepts:

- scopes — A scope specifies the area of interest for authorization within the kernel. For example, KAUTH_SCOPE_VNODE is useful for authorizing all events at the VFS layer. This makes it possible to register for authorization of only a subset of kernel events without having to deal with the rest.

  Scopes are strings in reverse DNS notation, for example, „com.apple.kauth.vnode" for KAUTH_SCOPE_VNODE scope, so it is also possible to define custom areas of interest.

- actions — An action is an operation within a scope. A combination of a scope and an action specifies the operations whose authorization will be checked. For example, KAUTH_VNODE_READ_DATA is an action of the VFS subsystem for reading data from a file system object.

- actors — Entity initiating the controlled operation.

- credentials — Credentials are information that identifies an actor. Kauth offers several access functions for working with credential objects, such as obtaining an EUID, GUID, and so on.

- request — An actor's request to perform an action within a scope. In order for a request to be allowed, no listener may return a deny decision. If all listeners return a defer decision, the request is denied.

- listener — A listener is a function invoked to perform authorization of an actor's request. The subsystem also contains a default listener for each built-in scope that implements an authorization model for all actions of that scope.

```c
static int MyListener(
    kauth_cred_t    credential,
    void *          idata,
    kauth_action_t  action,
    uintptr_t       arg0,
    uintptr_t       arg1,
    uintptr_t       arg2,
    uintptr_t       arg3
);
```

Listing 4.2: Prototype of Kauth Listener.

Listing 4.2 contains a prototype of Kauth listener. Meaning of the first three parameters is the same in all scopes:

[2] https://netbsd.gw.com/cgi-bin/man-cgi?kauth+9.i386+NetBSD-9.0

- `credentials` — a reference to actor's credentials,
- `idata` — the cookie supplied during the listener registration, and
- `action` — the requested action by the actor (i.e., `KAUTH_VNODE_READ_DATA`).

The meaning of the remaining parameters depends on the scope and usually contain extra information about the action (for example, a specific v-node number of the file that is being operated on). To finish the authorization process, a listener callback has to return one of the following values:

- `KAUTH_RESULT_DEFER` — indicates that the listener defers the decision to other listeners.
- `KAUTH_RESULT_ALLOW` — indicates that the listener allows the request.
- `KAUTH_RESULT_DENY` — indicates that the listener denies the request.

In case that all listeners return `KAUTH_RESULT_DEFER`, the request is denied. In order for the request to be allowed, at least one listener must return `KAUTH_RESULT_ALLOW`, and no listeners can return `KAUTH_RESULT_DENY` (which means that internally, the kernel has to notify all listeners about every request every time); otherwise, it is denied. `API` for listener registration and work with them is described in detail in [12].

Areas in which we are able to manage authorizations include:

- `KAUTH_SCOPE_PROCESS` — In the scope of processes, we can authorize process tracing (`KAUTH_PROCESS_CANTRACE`) and signaling a process (`KAUTH_PROCESS_CANSIGNAL`). However, the letter one was never implemented[3].
- `KAUTH_SCOPE_GENERIC` — The generic scope serves the kernel to test whether the actor has superuser privileges (`KAUTH_GENERIC_ISSUSER`) (equivalent to comparing `EUID` to 0).
- `KAUTH_SCOPE_FILEOP` — The file operations scope defines an action of opening a file system object `KAUTH_FILEOP_OPEN`, closing it (`KAUTH_FILEOP_CLOSE`), renaming it (`KAUTH_FILEOP_RENAME`), exchange the content of two files (`KAUTH_FILEOP_EXCHANGE`), addition of a new hard link to the file (`KAUTH_FILEOP_LINK`), and opening of the file for execution (`KAUTH_FILEOP_EXEC`). Unlike other scopes, this scope only alerts listeners to actions and ignores the return value of authorization. It is suitable for AV solutions, for example.
- `KAUTH_SCOPE_VNODE` — The v-node scope provides authorization of requests to work with files (read, write, execute, delete, ...), their attributes (e.g., timestamps), extended attributes, ACLs, and many others. This is the most complex scope of `Kauth`. The v-node scope, unlike other scopes, works with bit fields and not enumerations, so it is possible to authorize multiple requests at once.

The `Kauth` subsystem also has several possible pitfalls. In the case of authorization of v-node actions or file operations, these are very active areas (e.g., copying files can create thousands of requests per second), and inefficient implementation of the `Kauth` listener

[3] `www.opensource.apple.com/source/xnu/xnu-6153.61.1/bsd/kern/kern_authorization.c` :kauth_authorize_process_callback()

could cause a significant slowdown in file system operations. It may also happen that not all operations trigger an authorization request. For example, in the case of a request to search for an item in a folder, the system may cache the response and allow further requests without invoking the listeners [12].

Furthermore, in the case of `Kauth` authorization, these are blocking calls — the thread performing the operation creates the `Kauth` request, so the operation is blocked waiting for the result of the authorization. This can lead to cyclic dependency and deadlock. For example, by authorizing a file operation where the `Kauth` listener communicates with a user-space daemon to complete a decision, which calls some system routines opening a file (e.g., `lookupd`). This causes the kernel to call the listener again, and the cycle starts over. It should be noted that the dependency can also be caused indirectly. One solution would be to avoid dependency on system daemons, but this is not possible because a call can be caused, for example, by accessing a pageable memory, which might trigger the allocation of a paging file, which depends on the `dynamic_pager(8)`. Possible solutions are not to process requests coming from kernel threads (when `kauth_cred_getuid()` of credentials is 0). However, this technique may not be appropriate in all cases. For example, AV solutions that are particularly interested in scanning files being operated by a thread running with elevated privileges can operate entirely within the kernel [12]. A simple implementation of a `Kauth` listener that blocks operations within a specified folder based on [19] can be found in thesis' source files and its output in Listing D.3.

4.3 Mandatory Access Control

`Mandatory Access Control Framework (MACF)` is the result of a `SEDarwin` prototype[4] — a port of the `TrustedBSD MAC Framework` into `Darwin` which came with `Mac OS X 10.5`. Other implementations include `SELinux` and `AppArmor` for `Linux`, and `Solaris Trusted Extensions` for `Solaris`. Implemented interfaces are similar to the original `TrustedBSD` implementation but not the same (for example, different names of callbacks used) [67].

Unlike `Kauth`, which offers several areas of interest, the `MACF` provides access to important parts of the kernel and almost all system events, including file system operations, processes, memory protection changes, signature checks, and more. `MAC` unlike `Discretionary Access Control (DAC)`, which allows user/administrator to override security policies according to their preferences, `MAC` does not. Although `DAC` is usually sufficient for administrators, it is not enough for cases where also administrator account privileges need to be restricted. An example could be a `DLP` or `AV` solution, where even the administrator should not be allowed to easily control them (e.g., `Trojan Flashback.C OS X`, which took advantage of the fact that a regular `Mac OS X` user had pre-enabled escalation of privileges in the sudoers file and deleted the built-in Apple antivirus from system startup) [67]. `MAC` policies are not normally visible to users but are used in the background of services commonly encountered, such as system Sandbox or `System Integrity Protection (SIP)`. `MACF` is also used in the system to implement `Apple Mobile File Integrity (AMFI)` enforcing signature checking and mach port protection [80].

This framework is used in security software solutions despite the lack of support and documentation because of its irreplaceable power, and a wide range of operations that can be monitored and authorized. Whether to protect the user from malware or the program

[4] http://www.trustedbsd.org/sedarwin.html

```c
/**
 *  @brief MAC policy module registration routine
 *
 *  This function is called to register a~policy with the
 *  MAC framework.  A~policy module will typically call this from the
 *  Darwin KEXT registration routine.
 */
int mac_policy_register(struct mac_policy_conf *mpc, mac_policy_handle_t *hp, void *xd);
```

Listing 4.3: Prototype of MACF policy registration [22].

itself from unauthorized access, such as connecting a debugger by third-parties, signaling processes, or loading kernel extensions. It is also popular for malware development — the author of the `reverse.put.as` was probably the first known to use this framework to create a PoC rootkit `Rex` [40]. Unfortunately, `MACF` was never officially supported `KPI` by Apple, and its header files were included in the Kernel framework by mistake in `Mac OS X 10.5 SDK` [10]. This means that although Apple uses it internally, it is not documented nor recommended for use. The stability of the API (which, for example, varied between `Mavericks`, `Mountain Lion`, and `Yosemite`) is not guaranteed either [41]. Later, its header files were removed in `macOS 10.13 SDK` [45] (which they also warned against in the source file[5]). However, nothing is lost because the kernel loader and linker, `KXLD`, continues to load `KEXT`s using `MACF`, so it is possible to link the project against the older SDK and use it in the newer system [80]. Though, it is still necessary to keep in mind the instability of the API and its changes between versions, and in addition, the use of this framework could be a reason for rejecting the certificate request required to sign `KEXT`s [72, 51]. The possible individual hooks can be viewed in the `mac_policy_ops` structure in `security/mac_policy.h`.

```c
struct mac_policy_conf {
    const char                  *mpc_name;            /** policy name */
    const char                  *mpc_fullname;        /** full name */
    char const * const          *mpc_labelnames;      /** managed label namespaces */
    unsigned int                 mpc_labelname_count; /** number of managed label namespaces */
    const struct mac_policy_ops     *mpc_ops;         /** operation vector */
    int                          mpc_loadtime_flags;  /** load time flags */
    int                         *mpc_field_off;       /** label slot */
    int                          mpc_runtime_flags;   /** run time flags */
    mpc_t                        mpc_list;            /** List reference */
    void                        *mpc_data;            /** module data */
};
```

Listing 4.4: MACF policy configuration [22].

An example of tracking the execution of processes and their arguments can be found in Patrick Wardle's blog [72] or a more recent article by Fortinet [53]. Fortinet's blog also contains a detailed article about monitoring of file operations [54]. The concept of using `MACF` is simple. From the kernel extension, first register the `MAC` policy module by calling the `mac_policy_register()` function and then implement a callback that is invoked when performing the monitored operation. A prototype of the registration function can

[5] https://opensource.apple.com/source/xnu/xnu-6153.61.1/security/mac_policy.h.auto.html:85

be seen in Listing 4.3. The structure of the first argument, the configuration of the policy module, looks quite complex, but only four member variables are required to be filled in — unique policy name (ideally identical to the KEXT identifier), descriptive full policy name, a vector of operations for which the registered MAC policy has interest and flags of the policy (specifying whether the policy (and thus KEXT) can be unloaded and whether it must be registered during system boot) [53]. The complete structure of the policy configuration is in Listing 4.4. Because the vector of operations we have access to using MACF contains over 300 elements, Listing 4.5 contains only a selection of a few suitable operations for a DLP solution focused on disk and file operations. This structure is filled with pointers to callbacks that will be called for each operation and will be used in conjunction with the policy configuration during the registration, as shown in Listing 4.3.

```
/*
 * Policy module operations.
 *
 * Please note that this should be kept in sync with the~check assumptions
 * policy in bsd/kern/policy_check.c (policy_ops struct).
 */
#define MAC_POLICY_OPS_VERSION 58 /* inc when new reserved slots are taken */
struct mac_policy_ops {
    ...
    mpo_file_check_create_t                *mpo_file_check_create;
    mpo_file_check_dup_t                   *mpo_file_check_dup;
    ...
    mpo_mount_check_mount_t                *mpo_mount_check_mount;
    mpo_mount_check_remount_t              *mpo_mount_check_remount;
    mpo_mount_check_umount_t               *mpo_mount_check_umount;
    ...
    mpo_vnode_check_create_t               *mpo_vnode_check_create;
    mpo_vnode_check_read_t                 *mpo_vnode_check_read;
    mpo_vnode_check_write_t                *mpo_vnode_check_write;
    mpo_vnode_check_rename_t               *mpo_vnode_check_rename;
    mpo_vnode_check_unlink_t               *mpo_vnode_check_unlink;
    ...
}
```

Listing 4.5: Operation vector [22].

After the kernel module is loaded, each defined callback starts receiving notifications for particular event. An example is a callback for a file system mount operation, which can be seen in Listing 4.6. The most important for our use are kauth_cred_t (credentials that we already know from section 4.2), vnode, a structure of a v-node where the device will mount, and vfc_name, which is a type of file system to be mounted. As we can see, the callback authorizes a client's connection of a file system into a specific folder, but it does not provide an easy way to get more information about the connected disk.

For a more detailed description of TrustedBSD and its ports, including their differences, I recommend publications by its author Robert N. M. Watson [78, 79]. Mandatory Access Control parts in the macOS are explained, for example, in a presentation at Hack In the Box Security Conference by Jonathan Levin [50].

```c
typedef int mpo_mount_check_mount_t(
    kauth_cred_t cred,
    struct vnode *vp,
    struct label *vlabel,
    struct componentname *cnp,
    const char *vfc_name
    );
```

Listing 4.6: File system mount check callback prototype [22].

4.4 I/O Kit

As was mentioned at the beginning of this chapter, `I/O Kit` is an environment for driver development. A device driver can be used to implement a software that blocks external devices thanks to a way the driver matching works. When a device is connected, the `I/O Kit` searches for the most suitable driver, and in the last phase of this process, it can call the driver's `probe()` callback. In the callback, the driver can decide if it supports the device and returns a probe score to the system. In the end, the driver with the highest probe score is loaded. Using this technique, we can block devices of our interest by making a dummy driver to be loaded. The process of driver matching is described in more detail later in this section. This section is heavily based on information from [11, 18].

The `I/O Kit` abstracts the kernel capabilities, and hardware and provides a view to this abstraction to the upper layers of the operating system. Part of this abstraction is the implementation of behavior common to all types of device drivers in the classes of `I/O Kit`. This brings an advantage of a need only to add the specific code that makes the driver different and inherit common functionality that would need to be duplicated otherwise.

Every `I/O Kit` driver is based on an `I/O Kit family` that implements common functionality for all devices of the same type, such as storage devices, networking devices, or HID devices. `Provider objects` typically represent the bus connection used for communication with the device being controlled. They are also called `nubs`. A `nub` can also provide services to user-space through device interfaces, which is a library that can be used by an application in order to communicate with a device. Device interfaces and sending commands to devices from the user-space is described in section 6.1. The driver is loaded into the kernel automatically by `I/O Kit` when it matches against a device that is represented by `nub`. During the process of selecting a suitable driver, the system uses `personalities` in driver's `Info.plist` file defining types of devices the driver can control [11]. An example of an `I/O Kit` driver blocking all USB devices can be found in thesis sources and its loading attached in Listing D.4.

From developer's point of view, we have two ways of working with the `I/O Kit`. One is to work from the kernel using device drivers, and the second one is to communicate with device interfaces from the user-space. This section describes `I/O Kit` in general and device drivers that are to be resident in the kernel.

Architecture of the I/O Kit

The `I/O Kit` has a layered runtime architecture consisting of families, drivers, and nub objects; creating a dynamic stack of provider-client relationships among hardware and software involved in an I/O connection. Apple documentation perfectly describes the process of creation of a device tree mentioned in section 3.2 in one sentence: „The chain of interconnected services or devices starts with a computer's logic board (and the driver that controls it) and, through a process of discovery and "matching,„ extends the connection with layers of driver objects controlling the system buses (PCI, USB, and so on) and the individual devices and services attached to these buses." [18]. The current stack of devices can be showed using the `ioreg(8)` command or using `I/O Registry Explorer` utility provided in `Additional Tools for Xcode` package. A sample device tree for a Bluetooth driver can be seen in Figure C.1.

The I/O Registry and the I/O Catalog

The `I/O Registry` is a dynamic database capturing the client/provider relationship among active objects. This database tracks instantiated objects (drivers and nubs) and provides information about them. When hardware is added or removed from the system, the registry is immediately updated to reflect the current state of the system, which allows users to add devices to a running system and have them immediately available without the need for a reboot. A device driver must be recorded in the `I/O Registry` to participate in most `I/O Kit` services. The registry is accessible from user-space using the `I/O Kit` framework, which provides API for searching in the registry, retrieving a state of particular devices, and more.

At boot time, the `I/O Kit` registers a nub for the `Platform Expert`, which is a driver object for a particular motherboard that knows the type of platform the system is running on. This nub serves as a root of the `I/O Kit Registry` and is responsible for loading the correct platform driver which becomes the child node of the root. This driver then discovers the buses connected to the system and registers a nub for each one. The nub then loads and matches the most suitable driver. The process continues, and the `I/O Kit Registry` continues to grow. The registry resides in the system memory and is not stored or archived on a disk, thus created every time the system boots. The `I/O Kit Registry` can be examined using Apple's `I/O Registry Explorer` application or `ioreg(8)` command-line equivalent. Programmatically, properties of the registry can be explored and manipulated using member functions of the `IORegistryEntry` class.

The `I/O Catalog` is another dynamic database that maintains the collection of all `I/O Kit` classes (drivers) available on a system. When a nub discovers a device, it requests a list of all drivers of the device's class type from the `I/O Catalog`. This is the first step of a driver matching process.

Driver Matching

At boot time, and at any time a device is added or removed, the driver-matching process occurs for each detected device. Before a device can be used, its driver must be found and loaded into the kernel. As previously described, the matching process is triggered when a bus controller driver scans its bus and detects a new device attached to it. For each discovered device, a nub is created, and the `I/O Kit` starts the matching process. To support the matching process, each device driver must define one or more `I/O Kit` personalities in its information property list (`Info.plist`) specifying types of devices it supports. The matching process obtains information from the device (e.g., by examining the PCI registers) and dynamically locates suitable drivers in `/System/Library/Extensions` based on their information property list. When a nub detects a device, the `I/O Kit` finds the most suitable driver in three phases:

1. Class matching — this step eliminates drivers of the wrong device class.
2. Passive matching — this step eliminates drivers that do not match properties specific to the device, such as a vendor or product ID. All values of the driver's personality have to match a device in order to be selected for the device.
3. Active matching — if there are still any driver candidates left in the pool, the driver's `probe()` function is called (after the driver is instantiated) with reference to the `nub` it was matched against. This function allows the driver to communicate with the device and verify its support of it. The driver returns a probe score that reflects its ability to drive the device. During this phase, the `I/O Kit` loads and probes all candidate drivers and sorts them by the probe score.

The `I/O Kit` chooses the driver with the highest probe score. Typically, a more generic device driver loses out to the more specific drivers. After a matching driver is found, its code is loaded and an instance of the principal class listed in its personality is created. If the driver successfully starts, it is added to the `I/O Registry` and any remaining candidates are discarded, otherwise, the driver with the next highest probe score is started, and so on. A loaded driver, in turn, may create its own nub, which again initiates the matching process to find a suitable driver. A driver may also specify an initial probe score in its `Info.plist` and later in the `probe()` method change this default value based on its suitability to drive a device but its not recommended. More information about the matching process can be found in [18].

We can use the matching process to our advantage and write a generic USB driver that, in the passive matching phase, match against all devices of the USB class. This can be achieved by using a wildcard instead of a product ID and vendor ID. A driver with a '∗' instead of IDs matches with all the possible values. Later in the probing phase, we can inspect the device and decide if we are the „most suitable“ driver — this way, we can block selected devices. We can also decide if we ignore just `write()` calls to simulate some kind of read-only mode. This approach has several possible pitfalls, such as matching of our driver instead of drivers of a VM software's driver, or against a specialized device that needs an additional implementation in order to work even in read-only mode.

`I/O Kit` drivers can be also used as an alternative to generic kernel extensions. In this case, `IOProviderClass` is set to `IOResources`, which is a special nub connected to the root of the `I/O Registry`, and `IOMatchCategory` set to a private match value. This prevents the exclusive claiming of the `IOResources` nub, which would prevent other devices from matching on it. The value should be set to the driver's `IOClass` property, which is the driver's name in reverse-DNS notation with underbars instead of dots [18].

4.5 Summary

The operating system provides a number of options for monitoring and control of the system. Some of them are more focused on a smaller area, such as `IP/Socket Filters` or `I/O Kit`, while others provide a broader scope, such as `Kauth` or `MACF`. `Kauth` can be used to audit and authorize file operations (for example, blocking of write operations destinating in a cloud provider's folder). `Kauth` is (or rather was), unlike `MACF`, officially supported and documented stable API, but not so powerful. For process-tracking solutions, a problem can be that `Kauth` does not catch the `fork()` of a process [73], unimplemented feature of monitoring process signaling, inability to monitor `KEXT` loading, or other missing events in comparison with `MACF`. `MACF` is a very powerful framework providing many options to audit and control the system. However, it has never been supported by Apple for `KEXT` developers and comes with complications, such as its signature changes between `macOS` releases, and thus the need to check the system version at runtime, associated with removing framework's headers since `macOS 10.13 SDK`.

Kernel Extensions have advantages but also disadvantages in the form of its deprecation, and potential reductions in system performance and stability. Therefore, it is better to look for equivalent, such as System Extensions described in chapter 5 or user-space alternatives described in chapter 6. For the sole purpose of blocking devices, the use of either `KEXT` or `SYSX` could be a bit overkill as a much simpler and better solution is to use the `Disk Arbitration` framework described in the following chapter. On the other hand, if the implementation requires a hardware level of access, for example, to work with raw data in order to implement some kind of encryption, a suitable solution would be the new `DriverKit` framework based on the `I/O Kit`.

Chapter 5

Accessing System Using System Extensions

Information in this section is obtained mainly from Apple Worldwide Developers Conference (WWDC) [23] and Apple Developer Documentation [28, 30]. Kernel development is very complicated. Any issue with the kernel affects overall system operations, thus avoiding our code running in the kernel as much as possible brings more stability and security to the whole system. Released in October 2019, `macOS Catalina` introduced several significant changes and security features. Among the most important are System Extensions and `DriverKit`, which are substitution to kernel extensions used in previous versions of macOS.

Using a System Extension (`SYSX`)[1], an app can extend the operating system to be more reliable, more secure and easier to develop. A `SYSX` is a part of an app that extends the functionality of the operating system in ways similar to a kernel extension but running in the user-space, outside the kernel.

The biggest improvements of `SYSX`s over `KEXT`s are in the areas of security, privacy, and reliability. When a `KEXT` loads, it becomes part of the kernel. It has access to everything on the machine. This can be a danger because the kernel makes security rules (the kernel separates apps from each other and direct access to hardware, and allows them to use system services following the rules of security policy), the `KEXT` is above the rules. If a `KEXT` has a bug that allows it to be compromised, it can take over the entire machine. This means that any bug in a `KEXT`s can be a critical security problem. The System Extension runs in user-space. Like other apps, it has to follow the rules of the system security policy. Unlike other apps, `SYSX`s are granted special privileges to do specialized jobs. For example, they may have direct access to their associated hardware devices or use special APIs to communicate directly with kernel systems. If the SEXT crashes, the rest of the system and apps are unaffected and keep running. System Extensions are becoming the new standard, and future releases of `macOS` will not load `KEXT`s that have `SYSX` equivalent [23, 4].

System Extensions Layout

So far, there are three types of system extensions that can be built in macOS Catalina — `Network Extensions`, `Endpoint Security Extensions`, and `Driver Extensions`. `Network Extensions` are a replacement for `Network Kernel Extensions`. They can filter and re-route network traffic or connect to a VPN. For example, in the context of a DLP

[1] In various sources at the internet, `SEXT` is also used as an abbreviation to System Extension. However, `SYSX` is the term used during its introduction at WWDC 2019.

solution, one can prevent usage of AirDrop utility by blocking its ports or DNS queries using a Network Extension that runs entirely in user space. AirDrop utility after the creation of AWDL (Apple Wireless Direct Link) searches for `_airdrop._tcp.local` [83] and uses known ports[2][3]. Apple, unlike with other frameworks, provides a sample code which uses a `Network Extension` at the developer portal[4] [32].

`Endpoint Security Extensions` are replacement for `KEXTs` that intercept and monitor security-related events with the `Kauth` interfaces. It is suitable for appliances like Endpoint Detection and Response, Anti-virus, or Data Loss Prevention applications. `Endpoint Security` offers the possibility to audit and authorize actions like a creation of new processes, disk operations, and many more. A significant advantage in comparison with previous approaches, such as `OpenBSM` Subsystem is that `Endpoint Security` Framework provides the ability to proactively respond to system events. The `Endpoint Security` Subsystem provides a new preferred way of working with system events replacing previous ones, such as `OpenBSM` to track process', or `FSEvents` to track file operations.

The third type of an extension is `Driver Extension`, which is a replacement for device driver `KEXTs` using `I/O Kit`. In `Catalina`, one can control USB, Serial, NIC (Network Interface Controller), and HID (Human Interface Device) devices. `Driver Extensions` are built with `DriverKit` Framework. `DriverKit` is a new SDK with all-new frameworks based on `I/O Kit` , but updated and modernized; designed for building device drivers in the user-space. The kernel is an unforgiving and challenging environment to program within. The kernel is a conductor of everything that happens on the machine — it must never stop running, must never wait for anything to happen and must never crash. For example, kernel code has restrictions on where and how it can allocate memory or synchronize between threads, which means it cannot use most system frameworks, such as Foundation, since they are not designed to run in this environment. The only supported language for `KEXT` development is C and C++. System Extensions on the other hand have no such restrictions, which means they can be built with any framework in the `macOS` SDK and can use any language, including Swift. Though, because of `Driver Extensions`' close relation to hardware there are some restrictions. They must run in tailored runtime which isolates them from the rest of the system and must be written in C or C++. `SYSXs` are also easier to debug than `KEXTs`. Attaching a debugger to the kernel halts the kernel and entire machine, including the debugger. This usually means a two-machine setup is required. System Extensions, on the other hand, can be debugged while kernel keeps running. `SYSXs` can be built, tested, and debugged on one machine with full debugger support [23].

System Extensions Subsystem consists of several frameworks and libraries, system daemons, and kernel parts as can be seen in Figure 5.1. Assumably, from DLP developer's point of view, the most important one is `libEndpointSecurity.dylib` library, which is used to write `Endpoint Security` extensions. It registers itself as a client with an `Endpoint Security` kernel extension for listening to system events which then passes to the `Endpoint Security` Client. The kernel extension makes use of `MACF` and `Kauth` frameworks, which were described in chapter 4 to listen to system events.

Another interesting part is `SystemExtensions.framework`, which provides user application a way to install and uninstall system extensions. The rest of the subsystem is a group of system daemons and frameworks taking care of activating, deactivating particular types of System Extensions and running them which then communicate with the kernel. Elabo-

[2]Ports used by Apple products can be found at https://support.apple.com/en-us/HT202944

[3]Organizations can also restrict the use of AirDrop by using an MDM solution

[4]https://developer.apple.com/documentation/networkextension/filtering_network_traffic

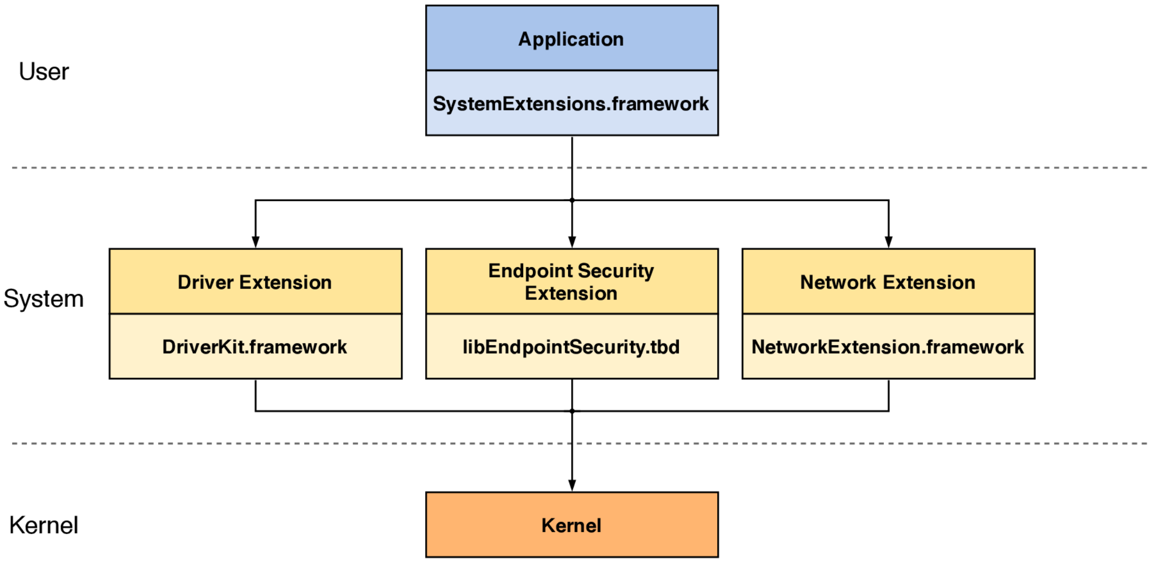

Figure 5.1: System Extensions Components [48]

rate scheme of `SYSX` internals can be found in Figure C.2 and information about the whole System Extensions Subsystem can be found in greater detail in great write-ups by Scott Knight [46, 47] and Jonathan Levin [51].

System Extension Installation

A System Extension is installed together with an application it is part of. The application uses a framework instead of an installer or a package. After a user (or the application itself) requests to activate a system extension using an API, the user has to approve the `SYSX` launch in System Preferences. After the `SYSX` is activated, the operating system takes care of the whole life-cycle of the `SYSX` as required.

`SYSX` are upgraded by updating the application that contains the extension. The operating system recognizes a new version and starts it instead of the old one. In order to uninstall the extension, the corresponding application just needs to be deleted. Moreover, applications can activate and deactivate an extension on-demand using System Extension Framework requests [48].

`SYSX` normally reside in `Contents/Library/SystemExtensions` folder of an application bundle. After a `SYSX` is activated and loaded by the operating system, it can usually be found at `/Library/SystemExtensions`. Activated System Extensions can be shown using command `systemextensionsctl list`.

Activation An extension is activated — as well as other actions with the extension — using `OSSystemExtensionManager` interface by sending an activation request. A System Extension is always part of an application. This is a basic principle of the design and there is no such thing as a standalone extension [23]. Listing 5.1 shows calls needed to activate a System Extension. During the activation process, the operating system verifies that the extension has proper entitlements to run and other checks depending on the extension type, namely:

- application bundle has to be installed in system's `Applications` directory,

- the extension has to be in the correct directory of the application itself,
- the code has to signed with an appropriate valid certificate (i.e., in case of `SYSX` it is a development team certificate),
- the entitlements in the code signature correspond to the application type and match the entitlements granted to the development team,
- identifier of the activation request and the System Extension match,
- and the identifier is not already in use by another `SYSX`.

In response, `macOS` calls a callback of the delegate with the result of the request. If the `SYSX` is being loaded for the first time, a user confirmation may be required. Moreover, a reboot of the computer may be required in order to deactivate and replace the running extension. A good description of all relevant classes and their methods can be found in [82].

```
// Create an~activation request and assign a~delegate to
// receive reports of success or failure.
let request = OSSystemExtensionRequest.activationRequest(
                        forExtensionWithIdentifier: driverID,
                        queue: DispatchQueue.main)
request.delegate = self

// Submit the~request to the~system.
OSSystemExtensionManager.shared.submitRequest(request)
```

Listing 5.1: System Extension Activation [30].

Update During the activation process, if the operating system detects an older version of the `SYSX` running, it passes the version information to both extensions and requests for action. Installation can be either canceled, or the extension replaced by the system[5].

Uninstall An extension is deactivated automatically together with removal of the corresponding application. However, it can also be done by the application by sending a request to the `OSSystemExtensionManager`.

5.1 Network Extensions

`Network Extensions` are one of APIs under the umbrella of the System Extensions. It allows developers to inspect and filter network traffic, write DNS proxies, and VPN clients. Using `Network Extensions`, one can customize and extend system networking features and implement functionality like on-device content filter, on-device DNS proxy, creating and managing VPN configurations, or setting the system's Wi-Fi configuration. This section is mainly based on information from [32].

A content filtering ability of the `Network Extensions` provides a way to examine user network content as it passes through the network stack. User's privacy is taken into account,

[5]developer.apple.com/documentation/systemextensions/ossystemextensionrequestdelegate/3295277-request

and the content filtering code used with `Network Extensions` API runs in a very restrictive sandbox and does not allow filtered content to escape the sandbox. The code controlling the filter runs in a separate, less restrictive sandbox and passes configuration information to the filter. It does not have access to network data. Using this separation, the filter has access to the network content but cannot export the data out of its sandbox. At the time of writing this thesis, the framework provides two content filter providers — `NEFilterPacketProvider`, providing the ability to evaluate network packets and decide whether to block them or not, and `NEFilterDataProvider`, working with raw data. A DNS proxy provider implements DNS proxying and takes responsibility for resolving DNS queries on the system. It can be used to filter or forward DNS queries, or implement a custom DNS proxying protocol [32]. A sample implementation of a project using `Network Extensions` can be found in Apple's documentation[6].

5.2 Driver Extensions

`DriverKit` is a new framework that extends `I/O Kit` SDK. Since `macOS Catalina`, `DriverKit` should be a preferred way of developing drivers as `I/O Kit` drivers of device families included in the `DriverKit` will not be loaded in the future. Driver Extensions (`DEXT`) SDK has a limited API surface for reliability and security, and there is no direct access to the file system, networking, and mach messaging. This allows Apple to tailor an user-space process to running drivers, and to give it an elevated priority and increased capabilities. Drivers developed using `DriverKit` use a new `.dext` extension and can be located at `/System/Library/DriverExtensions` and `/Library/DriverExtensions`. Although `DEXT` run completely in the user-space, which also allows using of system frameworks compared to `KEXT`s, there is no `LC_MAIN` command in the executable's Mach-O header, which means it cannot be run directly using terminal or `lldb` [23].

`DriverKit` also comes with a new filetype, `.iig`. It defines C++ class interfaces and is processed by the `I/O Kit Interface Generator` (`IIG`) tool. The tool is a replacement for `Mach Interface Generator` (`MIG`) in `DriverKit`, which autogenerates C++ code.

When a new device appears that has a driver extension, the `I/O Kit` creates a kernel service to represent driver's service, and a new process hosting the driver within an instantiated `DriverKit` class is created. The process also has proxy objects for any services it uses, such as its provider. This can be seen in Figure 5.2 where the `DriverKit USB Device` is a proxy object that represents the `USB Device` object in the kernel. Thanks to the proxy object, it appears to the kernel extensions and can compete in matching with kernel drivers. They can be seen in the `I/O Kit Registry` as well, and the `I/O Kit Framework` API can be used with them. Each instantiated `DriverKit` driver has its own process which is separated from the kernel and other drivers [23].

In order to distribute a Driver Extension, developers need to obtain several entitlements from Apple. There is one for all `DEXT`s — `com.apple.developer.driverkit`, and one to take control of a device — `com.apple.developer.driverkit.transport.usb`. There is also a family entitlement to make an available service to the OS [23].

[6] https://developer.apple.com/documentation/networkextension/filtering_network_traffic

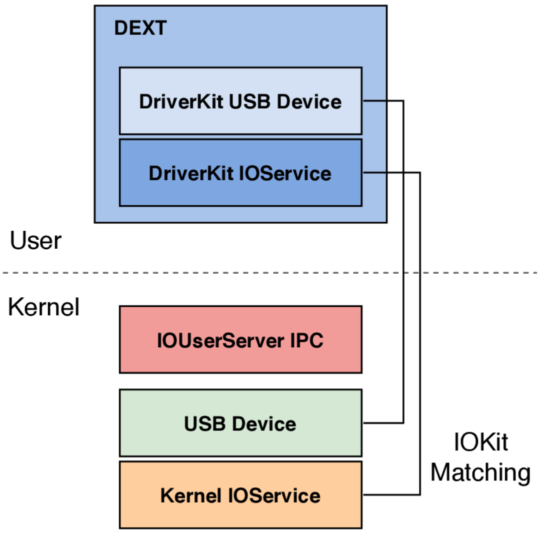

Figure 5.2: I/O Kit Matching [23]

5.3 Endpoint Security

To begin developing in `Endpoint Security`, its client needs to be created. This can be done with `es_new_client()` call. The client is then used to subscribe to event types of interest. When `Endpoint Security` monitors an event, it sends a message describing the event to clients that subscribed to a particular event. When the client is no longer needed, it can be destroyed by the `es_delete_client()` call. Listing 5.2 describes the creation of a sample client that subscribes to authorize the opening of a file with the „dropbox“ string in its path or name. Once the subscribe function successfully returns, the client will start receiving events from the `Endpoint Security` Subsystem [25].

Each security event being processed by the `Endpoint Security` subsystem is encoded in an `es_message_t` object. When an event occurs, `Endpoint Security` sends the message to every client that subscribed to the event type. The content of the message is listed in Listing 5.3. The message contains additional data about the event like source and destination paths during rename operation, process identifier of the new child during process fork, etc. There are two message types — an authorization request and a notification of an event that has already taken place [31].

There are two main event types: `ES_EVENT_TYPE_AUTH_`* and `ES_EVENT_TYPE_NOTIFY_`*. Data needed to differentiate various events are stored in `action_type` field of the `es_message_t` object (i.e., if the message is just a notification event or an authorization request) and `event_type` property, which indicates the exact type of the operation (and obviously is one of the events we subscribed to) [31]. Event types which are interesting for a DLP solution handling devices are listed in Table 5.1.

Action type is useful to determine if the message has to be processed immediately or can be deferred and processed later. While the notification action type contains just an opaque event identifier and requires no action, the authorization action type requires

[7] Taken from `ESMessage.h` header file of Endpoint Security framework owned by Apple Inc, which can be found at usr/include/EndpointSecurity in the MacOSX10.15.sdk.

```
// Create the~handler block run on all messages sent to this client
es_handler_block_t handler =
        ^void  (es_client_t* _Nonnull client, const es_message_t* _Nonnull message) {
    switch (message->event_type)
    {
        case ES_EVENT_TYPE_AUTH_OPEN:
        {
            es_auth_result_t res =
                strstr(message->event.open.file->path.data, "dropbox")
                ? ES_AUTH_RESULT_DENY : ES_AUTH_RESULT_ALLOW;
            es_respond_auth_result(client, message, res, false);
            break;
        }
        // case: (Handle any other cases you have subscribed to)
        default:
            break;
    }
};

// Create the~client.
es_client_t *client = nullptr;
es_new_client_result_t client_result = es_new_client(&client, handler);
// Handle any errors encountered while creating the~client.
...
// Subscribe the~client to the~ES_EVENT_TYPE_AUTH_OPEN event.
// When the~client receives a~message with this event type, it must authorize
// (allow or deny) the~event.
es_event_type_t eventTypes[] = { ES_EVENT_TYPE_AUTH_OPEN };
es_return_t subscribeResult = es_subscribe(client, eventTypes, sizeof(eventTypes));
if (subscribeResult != ES_RETURN_SUCCESS)
    panic ("Client failed to subscribe to event.");
```

Listing 5.2: Endpoint Security Client Example [25].

```
typedef struct {
    uint32_t version;
    struct timespec time;
    uint64_t mach_time;
    uint64_t deadline;
    es_process_t * _Nonnull process;
    uint64_t seq_num; /* field available iff message version >= 2 */
    es_action_type_t action_type;
    union {
        es_event_id_t auth;
        es_result_t notify;
    } action;
    es_event_type_t event_type;
    es_events_t event;
    uint64_t opaque[]; /* Opaque data that must not be accessed directly */
} es_message_t;
```

Listing 5.3: es_message_t structure[7].

Table 5.1: Selected Endpoint Security Events [29].

`ES_EVENT_TYPE_AUTH_MOUNT`	Authorizing the mounting of a file system.
`ES_EVENT_TYPE_AUTH_OPEN`	Authorizing the opening of a file.
`ES_EVENT_TYPE_AUTH_READDIR`	Authorizing the reading of a file-system directory.
`ES_EVENT_TYPE_AUTH_RENAME`	Authorizing the renaming of a file.
`ES_EVENT_TYPE_AUTH_UNLINK`	Authorizing the deletion of a file.
`ES_EVENT_TYPE_NOTIFY_WRITE`	Notification of the writing of data to a file.

a response [24]. The client must respond before a deadline specified in the message. If it fails to respond in time, Endpoint Security may kill the client process, or restart it in case of a System Extension. If it fails repeatedly, Endpoint Security may refuse further connections from the client [27].

In the case of a notification message, a client gets information about type of the event that happened and result of an action that has taken place. For example, the `ES_EVENT_TYPE_NOTIFY_OPEN` notifies the subscriber that a file has been opened, and the result of the action contains flags that were used to open the file. For authorization events, the system requires a response before being able to proceed. In the case of the `ES_EVENT_TYPE_NOTIFY_OPEN` event, the action will block until the `Endpoint Security` subsystem client responds or until deadline. The action of the authorization message contains an opaque authentication ID that must be supplied when responding to the event [76, 75].

Some of the events require authorization flags as a response (i.e., `ES_EVENT_TYPE_AUTH_OPEN`) and some allowing or denying value (all other authorization event types). The result is stored in `es_result_t` structure shown in Listing 5.4. The structure is only in messages of notification events and indicates the result of the `Endpoint Security` subsystem authorization process.

```
typedef struct {
    es_result_type_t result_type;
    union {
        es_auth_result_t auth;
        uint32_t flags;
        uint8_t reserved[32];
    } result;
} es_result_t;
```

Listing 5.4: es_result_t structure[8]

All file events messages (as well as other events) contain a pointer to a `es_process_t` structure. Using this structure, one can identify the responsible process that generated the event, including its PID, path, code signature. This can be used if the program should allow the operation or not. For example, it can only allow read-write operations of selected processes. Chapter 8 discusses how to utilize this framework in order to provide a user read-only access into a local cloud drive but to allow all cloud services, such as synchronization with the remote site. A simple implementation of an `Endpoint Security` client can be

[8]Taken from `ESMessage.h` header file of Endpoint Security framework owned by Apple Inc, which can be found at usr/include/EndpointSecurity in the MacOSX10.15.sdk.

found attached with the thesis sources and its output in Listing D.6. The demo is based on a code from Omar Ikram[9].

5.4 Summary

System Extensions provide an environment to avoid difficulties and dangers of kernel programming by running in user-space. In future versions of macOS, more kinds of `SYSX`s and `DriverKit` device families will be added, and in turn, `KEXT`s of those kinds will be deprecated.

Before, writing a utility monitoring system events was not a trivial task. For a software solution that needed to monitor and apply a policy to events originated by other processes and applications, possibly also running under different user accounts or even the operating system, there was no user-space solution, and the kernel was the only place that gave the level of required access. Now, thanks to the `Endpoint Security` Framework it is much more straightforward. In comparison to `Kauth` and `MACF` described in chapter 4, `ESF` provides rich information regarding various system events that would have to be retrieved using unsupported-kernel techniques otherwise. System Extensions subsystem provides great possibilities entirely from user-space. `SYSX`s are expected to continue expanding over the time as it is still under development (as can be seen by frequent changes in API and its documentation).

[9] `https://gist.github.com/Omar-Ikram/8e6721d8e83a3da69b31d4c2612a68ba/`

Chapter 6

Accessing System From User-Space

This chapter describes various ways of monitoring system events from user-space, without the need of any special certificates or entitlements that were needed in order to use `KEXT`s and `SYSX`s. Some of the frameworks and facilities described in this chapter offer only monitoring capabilities, however, as you will see in the following sections, information provided by these frameworks might be essential to decide if to block particular action or not.

6.1 I/O Kit

Because of kernel-space and user-space address space separation, normally, user-space applications cannot access kernel's address space. However, some programs need to control or configure a device, thus need access to `I/O Kit` services in the kernel (for example, a game setting monitor depth, scanner applications, and so on). To fulfill this requirement, the `I/O Kit` includes two mechanisms: device interfaces and `POSIX` device files. This section is mainly based on [18, 9]. A sample implementation of an application that monitors external devices being connected is attached with the thesis sources.

Device Matching

Device matching is the process of searching for an `I/O Kit` object representing a specific device or device type. Unlike driver matching, device matching searches for a driver that is already loaded. For example, an application running in user-space can initiate a search for all `I/O Registry` objects representing USB storage devices. The search is based on a matching dictionary that contains key-value pairs specifying the properties of the object that is searched for. These properties can be shown by `ioreg(8)` utility or `I/O Registry Explorer` application.

Device Interfaces

As mentioned in section 4.4, a nub provides device interfaces to allow user-space applications to communicate with devices through the kernel. A user-space program can communicate with a nub in the kernel that is appropriate to the type of device it wishes to control through plug-in architecture and defined device interfaces.

On the user-space side, the device interface enables communication with the application through its exported programmatic interfaces. On the kernel side, it enables communication with an appropriate `I/O Kit` family through a nub created by the driver object of that family. From the kernel's perspective, a device interface appears to be a driver and is known as a `user client`. Stack created after the acquisition of the `IOFireWireDeviceInterface` provided by the `IOFireWire` family is shown in Figure 6.1. From the application's point of view, the interface is a set of functions that can be used to pass data to the kernel and receive it back from it. At an elemental level, the device interface is usually a pointer to a function pointers table and the application can call any of the interface functions [18].

Both the user client and the device interface are provided by the `I/O Kit` device family. Not all device interfaces are exported to the user-space, though. Some of them have to be accessed from inside the kernel because of security and stability reasons, e.g., PCI family interfaces. Exported interfaces can be found in the Apple documentation [18]. When an application requests a device interface of a particular device, the device family instantiates a user client object and, typically, attaches it in the `I/O Registry` as a client of the device nub. An application that acquired the device interface can act as a user-space driver for that device. Communication with the `I/O Kit` uses Mach ports introduced in section 3.2.1, but apart from getting the master port of the `I/O Kit` at the beginning, there is pretty much nothing more to be done with them [9].

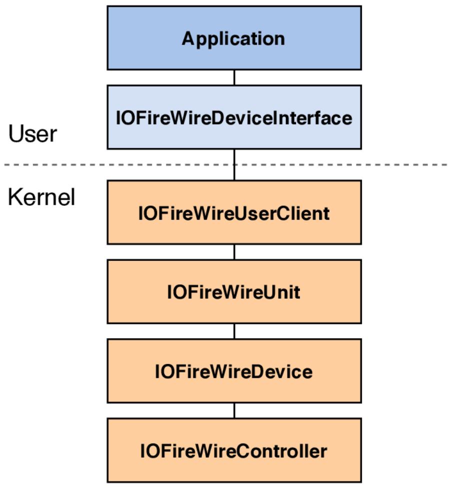

Figure 6.1: Device interface in a FireWire driver stack [9]

Device Files

The `BSD` part of the kernel exports number of programmatic interfaces that are consistent with the `POSIX` standard. These interfaces enable communication with serial, storage and network devices through special files located in `/dev` directory. These files represent block and character devices. Files can be then used by `POSIX` functions such as lstinlineopen(2), `read(2)`, `write(2)`, and `close(2)`. These files are dynamically created after the `I/O Kit` discovers devices. These files can change disk they are representing over the time, and the same disk can have different device-file names at different times, so it's recommended to get the device `BSD` path from the `I/O Kit` involving device matching [9].

One might think that a possible way of blocking access to external devices would be by changing read/write permissions to the `/dev` file. This is not the case because, first, in order to work with `BSD` device files of, for example, a USB drive (i.e., to open them), the drive has to be unmounted first but still connected with driver loaded. Then, even by changing `read/write` file system permissions of the block device, the user program still can access the device without any change. The only way to effectively block access to the device was to exclusively open the block device file. However, this is not an acceptable approach as all other applications are denied access to the device or folder with no control over it.

6.2 DTrace Probes

`DTrace` is a framework that provides capabilities to debug the kernel and applications in a production environment in real-time (thus with the least possible impact on system performance). It was originally developed by `Sun Microsystems` for `Solaris 10` and subsequently ported to other systems, such as `Linux`, `FreeBSD`, `Windows 10`[1], and `macOS`[2].

`DTrace` can be used to get a general overview about a running system, such as memory usage, CPU, file system events, and network resource usage, but also for detailed monitoring, including arguments to specific functions called, a list of processes accessing a particular file, and so on. `DTrace` works with programs (and scripts) in `D` programming language (similar to `C` with its structure)[3]. `macOS` contains a prepared collection of `D` scripts, `DTraceToolkit`, which contains utilities like `/usr/bin/iosnoop` monitoring disk I/O operations, or `/usr/bin/execsnoop` which monitors the execution of processes in the system. Other scripts can be found, for example, by calling `apropos DTrace` command.

Scripts connect from user-space to points called probes. The user can use the probe to display relevant kernel or process information from user-space. Each probe is activated by a specific action, such as entering a kernel function. The particular probe can display additional information, such as arguments passed to the function, global kernel variables, event timestamps, the current stack trace, the thread and process that called the function, and more. We can specify an action that the probe does when it gets activated; for example, it can simply record the event. For more on the capabilities of the `DTrace` framework, I recommend the `DTrace User Guide` from `Oracle` [60].

In addition to other platforms, `macOS` supports the `P_LNOATTACH` flag, which the program sets if it wants to disable its monitoring by debug utilities such as `DTrace` or `gdb` (although it can still be bypassed [56]). As of `OS X 10.11 El Capitan`, `DTrace` options are limited by `SIP` protection (among other things, also to protect private data), and system tools such as `execsnoop` do not work either [72].

6.3 Kernel Debug

The `macOS` kernel provides the ability to track its events using `kdebug`. This facility is also used by the system tools `sc_usage(1)` (monitoring system calls and various page faults), `fs_usage(1)` (similar to `sc_usage(1)` limited to file system operations), and `latency(1)` (monitors scheduler and interrupt latency statistics). Among other things, it can also be used as a source of entropy to generate (pseudo) random numbers. It is a powerful but poorly documented tool in both `macOS` and `iOS`, which is disabled by default but can be enabled by `sysctl(8)` call [64]. Applications can also use the `kdebug` interface to track their own events.

[1] https://docs.microsoft.com/en-us/windows-hardware/drivers/devtest/dtrace
[2] https://support.apple.com/kb/SP517?locale=en_US
[3] `https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-924.pdf`

`Kdebug` categorizes monitored operations into classes, class subclasses, and subclass codes. In addition, some kernel functions mark the entry with a start and end flag (`DBG_FUNC_START`, `DBG_FUNC_END`, or `DBG_FUNC_NONE` in case of non-function records). Because `kdebug` uses for records kernel buffers and their space is severely limited, individual classifiers defined in `bsd/sys/kdebug.h` are squashed into a single 32-it code, which is then used in the record. The header file also contains helper macros for creating and parsing the classifiers. The hierarchy of classifiers can be seen in Figure 6.2 [49].

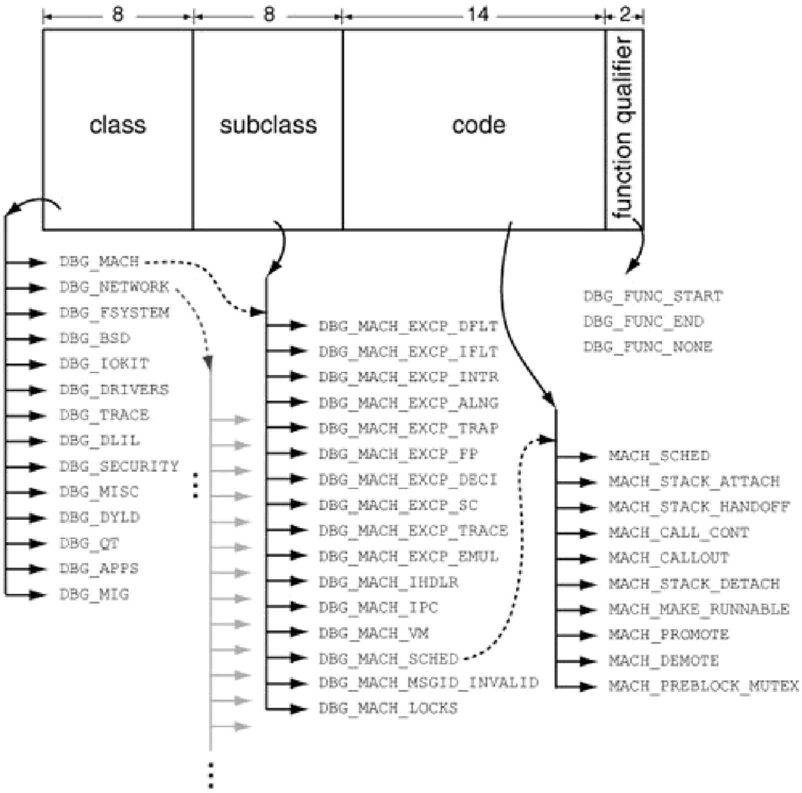

Figure 6.2: Kdebug Classifiers Hierarchy [64]

```c
int set_kdebug_enable(int value) {
    int rc;
    int mib[4];

    mib[0] = CTL_KERN;          mib[1] = KERN_KDEBUG;
    mib[2] = KERN_KDENABLE;     mib[3] = value;
    if ((rc = sysctl(mib, 4, NULL, &oldlen, NULL, 0) < 0)
    {perror("sysctl");}
    return (rc);
}
```

Listing 6.1: Enabling of kdebug [49]

`Kdebug` is accessed from user-space using `KERN_KDEBUG` sysctl operations with `CTL_KERN` identifier at the highest sysctl level. For enabling, `kdebug_enable` has to be set to a nonzero value, and although the variable is not visible from user-space, we can set it using a `sysctl(2)` call, as is shown in Listing 6.1. Supported operations include:

- enabling or disabling of tracking (`KERN_KDENABLE`),
- clearing up the relevant trace buffers (`KERN_KDREMOVE`),
- tracing system reinitialization (`KERN_KDSETUP`),
- specifying the trace buffer size (`KERN_KDSETBUF`),
- specifying which PID to trace (`KERN_KDPIDTR`),
- specifying which PID to exclude (`KERN_KDPIDEX`),
- specifying trace points of interest based on classifiers or a range of debug code values (`KERN_KDSETREG`),
- retrieving trace buffer metainformation (`KERN_KDGETBUF`), and
- retrieving a trace buffer (`KERN_KDREADTR`) [64].

The disadvantage is that `kdebug` can only be used by one program at a time, which is not very useful for continuous system monitoring. Another one is that although we can track a vast number of operations, apart from the fact that the operation itself occurred, we have very little extra information.

Kdebug Internals

The kernel contains special `KERNEL_DEBUG_CONSTANT` and `KERNEL_DEBUG_CONSTANT1` macros shown in Listing 6.2 in various places of the source code. These macros allow one to track events, such as system calls, mach traps, file system operations, or `I/O Kit` traces. The `kernel_debug1()` is a special version of `kernel_debug()` which is used in the case of a `execve()` call followed by a `vfork()` call. Function `kernel_debug()` uses `current_thread()` function to determine the identity of a thread, which in the case of the mentioned chain of calls returns the identity of the parent thread. That's why `kernel_debug1()` is used, and data needed for process identification are passed to it in arguments. For more detailed information, I recommend a book from Amit Singh: Mac OS X Internals: A Systems Approach [64]. The book was also a source of demo program attached with thesis sources, and its output can be found in Listing D.7.

6.4 Kernel Events

Kernel Queues (`kqueue`) and Kernel Events (`kevent`) are a flexible mechanism introduced in `FreeBSD 4.1` and later ported to `Mac OS X 10.3 Panther`, allowing to receive kernel-level events with the possibility of filtering only events of interest [14, 35].

An application works in the user-space where it registers events for monitoring, the kernel stores them in the queue and then returns them when the queue is queried. Kernel queues currently support monitoring of file descriptor events, processes, signals, asynchronous I/O operations, and v-nodes. Kernel queue can be known from `FreeBSD` where it

```
// bsd/sys/kdebug.h
#define KERNEL_DEBUG_CONSTANT(x,a,b,c,d,e)  \
do {                                        \
    if (kdebug_enable)                      \
        kernel_debug(x,a,b,c,d,e);          \
} while(0)
#define KERNEL_DEBUG_CONSTANT1(x,a,b,c,d,e) \
do {                                        \
    if (kdebug_enable)                      \
        kernel_debug1(x,a,b,c,d,e);         \
} while(0) ...
```

Listing 6.2: Tracing macros

comes from[4]. In the `macOS` man pages, it can be seen that the Apple version of `kqueue(2)` is not fully identical to `FreeBSD`[5] version where, for example, filters for opening and closing a file are missing, while events for mach ports are added. This mechanism brings considerable overhead caused by forwarding notifications from kernel to the user-space.

Among other things, the interface provides options for monitoring events related to processes, such as their termination, fork, execution, and signal reception. File-related events that can be monitored are listed in Listing 6.3. The disadvantage is that the monitoring is started for selected `PID` or file descriptors, respectively. This could be suitable for monitoring specific processes or files (such as ours), but it is not suitable for a system-wide event monitoring.

To monitor files, first, a kernel queue must be created by calling `kqueue()`. Then, files that are about to be monitored need to be open and used when setting up the `kevent` structure containing filters. Finally, `kevent()` is called in a loop to check for events added to the queue. An example of an implementation that tracks changes to a specific file can be found in source files of the thesis and its output in Listing D.8. The implementation is based on [14].

```
// from sys/event.h
#define NOTE_DELETE     0x00000001      /* vnode was removed */
#define NOTE_WRITE      0x00000002      /* data contents changed */
#define NOTE_EXTEND     0x00000004      /* size increased */
#define NOTE_ATTRIB     0x00000008      /* attributes changed */
#define NOTE_LINK       0x00000010      /* link count changed */
#define NOTE_RENAME     0x00000020      /* vnode was renamed */
#define NOTE_REVOKE     0x00000040      /* vnode access was revoked */
#define NOTE_NONE       0x00000080      /* No specific vnode event: to test for
                                           EVFILT_READ activation*/
#define NOTE_FUNLOCK    0x00000100      /* vnode was unlocked by flock(2) */
```

Listing 6.3: V-nodes events.

[4] https://people.freebsd.org/~jmg/kq.html
[5] https://www.freebsd.org/cgi/man.cgi?query=kqueue&apropos=0&sektion=0&format=html

6.5 File System Events

File System Events (FSEvents) API allows applications to register of receiving notifications of changes to selected file system subtree. When there is a change in the directory structure of the monitored folder, the client receives a notification. It also allows us to get all changes from the past using a timestamp or event identifier [5]. Information in this section is drawn mainly from the FSEvents documentation [14].

In the documentation itself is written that this technology is not intended for detailed tracking of file system changes, thus it is not suitable for AV or DLP solutions or other software requiring immediate notification about the change. The documentation recommends a kernel extension that registers for interests in changes at the VFS level [14]. FSEvents API is suitable for passive tracking of changes in large file system subtrees, which is useful, for example, for backup software such as Apple's Time Machine. Another way to use the FSEvents facility is to work directly with a special device used by the framework itself. Both the use of the special device and the FSEvents API access information from user-space and need elevated privileges.

FSEvents API

The FSEvents API consists of two main groups of functions — one starting with FSEvents and the other starting with FSEventStream. The former one is used to obtain information about disks and events, and latter to work with an event stream. To receive notifications, first a stream object has to be created with the FSEventStreamCreate() call or FSEventStreamCreateRelativeToDevice() call for tracking notifications per-device. Next, it needs to be classically scheduled in Run Loop, and then FSEventStreamStart() called in order to start receiving notifications from the system daemon. To end the subscription, use FSEventStreamStop(), disconnect it from the Run Loop, and invalidate and release the stream object [14].

The registered callback receives three lists representing a list of events — a list of changed paths, a list of identifiers, and a list of tags. For each event, the specified path should be checked after the callback is invoked and the content processed according to the tags (e.g., recursive path processing, event drop notification due to full buffers, etc.). Tags also include kFSEventStreamEventFlagUnmount and kFSEventStreamEventFlagMount[6], so it is easy to distinguish disk mounts from normal file operations.

From a security perspective, it would not be appropriate for a regular user using this API to receive system-wide notifications. This could lead to leaks of sensitive data even just from the names of the files being worked with. Therefore, the user only receives notifications from folders to which he/she has access based on standard file system permissions. Obviously, only a program running with root privileges can be guaranteed to receive all notifications.

As can be seen from the output in Listing D.9, it is missing quite substantial information, namely the PID of the process that performed the action, and also its timestamp. These information are also missing in .fseventsd folder which contains the entries used (also) by this API [57]. This API is therefore suitable, for example, for backup software which can get either information about changed folders or specific types of operations, and it can also retrieve this information from the past. However, for a DLP solution, there is too little information provided to decide on the eligibility of the operation.

[6]Defined in the FSEvents.framework/Versions/A/Headers/FSEvents.h of CoreServices.framework in the macOS SDK.

Direct Access to the Events Device

The second way to use the `FSEvents` facility without using its API is to directly open the special block character device used by the framework, `/dev/fsevents`. Although this way we lose the ability to retrieve events from the past, for a DLP solution that ideally runs continuously, this should not be a problem. In this way, we are able to obtain more detailed information than using the API. For a DLP solution, perhaps the most important benefit is that we have the `PID` of the process that made a change in the file system, as can be seen in the sample output attached in Listing D.10.

I need to recall that this approach is not supported by Apple[7] and its interface, including the structures used, may vary from version to version. Other disadvantages include the already mentioned ability to receive events only from client startup and not from the past and the need to (painfully) parse and process raw (binary) event data, as can be seen in the attached sample code attached with the thesis. Security solutions may not like that only the `PID` is known and not something more unique. `PID` is reused over the time, and there is also a delay between getting the event and the detection of additional data to the `PID`, like the process path (which may no longer exist). This can lead to incorrect process identification, which is not acceptable for a security software. More about issues with `PID` can be found in a talk by Samuel Groß at WarCon security conference [44]. By working directly with the `/dev/fsevents` device, we get events with higher granularity, and cloning the `/dev/fsevents` file gives us the ability to have multiple clients at once. However, during the cloning, the size of the read event queue (`event_queue_depth`) needs to be chosen carefully so it is not too large, which could cause dropping of events by other (system) daemons, such as `mds` [74]. A sample implementation of both approaches can be found in the thesis' source files which are based on code provided by [64, 49].

FSEvents Internals

The predecessor of the File System Events `FSEvents` API was added in `Mac OS X 10.4 Tiger`. It was a private API used by `spotlight` to track changes to the file system and index files. Jonathan Levin was possibly the first to use it in his `fslogger`[8] tool showing the potential of the interface. In the description of the tool, he pointed out the risks of using this private interface, such as its reservation for `spotlight` and the possibility of causing the event buffer to overflow by adding another interface clients. A more detailed description of possible problems and the history of the `FSEvents` API can be found in a detailed article by Ars Technica [66]. The special `/dev` file allowed `spotlight` to receive notifications about new files and their changes without the need for polling. However, this also means that any program that would like to receive notifications from `/dev/fsevents` would have to run from the system startup for the whole time and receive all notifications, which can be a burden on the system for several such clients.

`FSEvents` API was added to a later version of `Mac OS X 10.5 Leopard`. Until `Mac OS X 10.7 Lion` it was not possible to track individual files, but only selected directory. This has been changed in `Mac OS X 10.7 SDK` which added support for notification of changes to individual files in the monitored file system subtree[9].

When there is a change within a file system, the kernel sends a notification via the special device `/dev/fsevents` to the user-space process `fseventsd` which reads from this file and

[7] `https://opensource.apple.com/source/xnu/xnu-6153.61.1/bsd/vfs/vfs_fsevents.c`:add_watcher()
[8] http://osxbook.com/software/fslogger/
[9] Using the `kFSEventStreamCreateFlagFileEvents` flag

logs events into a compressed binary file stored in the `.fseventsd` folder in the root of the monitored disk. This process aggregates events from a single folder that occurred in a short period of time into a single log and notifies clients that have registered to track the folder. These may not be running all the time but may ask about changes from a point in time in the past. This has the advantage of recognizing changes made in another operating system (i.e., in case of USB drives) that `/dev/fseventsd` does not know about because it happened outside of the current kernel. Further reading about the use of the framework by `macOS` components can be found in an Howard Oakley's article [57]

6.6 Disk Arbitration

`Disk Arbitration` is a framework that can be used by user-space applications to receive notifications about connected disks and allows them to influence the process of mounting new devices. In particular, it allows the developer to detect the discovery and connection of a new disk, prevent the connection of a given disk, disconnect a disk and connect it with added or changed parameters, and monitor changes in device names. To perform the function of an arbitrator, functions are available to obtain more information about a given device, such as the `I/O Register` object, `BSD` name, description, manufacturer and device ID, and more. A good thing about the `Disk Arbitration` is that it runs in user-space. That means that to take over the control of external devices, it is enough just to run an application and no more special procedures need to be done. The important fact is that the `Disk Arbitration` cannot control hardware ports or external devices that are not volumes. This section is mainly based on [16].

`Disk Arbitration` functions can be divided into two groups — notification and authorization. For example, notification ones can be used for the detection of a new external device to offer it for use as a backup storage. Notification callbacks include:

- `DADiskAppearedCallback` – called when a disk appears, or a partition appears,
- `DADiskDescriptionChangedCallback` – called when a disk's description has changed (and, in OS X v10.7 and later, when a volume is first mounted),
- `DADiskDisappearedCallback` – called when a removable disk is ejected, and
- `DADiskPeekCallback` – called when a disk is first probed, before automatic mounting begins, and before any other notifications are sent out.

Authorization callbacks can be used to influence the behavior of external device connection process. Authorization functions include:

1. `DADiskEjectApprovalCallback` – called when a disk is about to be ejected. Eject operation is usually going to be called alongside unmount, therefore, unmount callback is more general and will catch more cases.
2. `DADiskMountApprovalCallback` – called when a disk is about to be mounted. This provides a functional alternative to device control because we can make sure that a user cannot leak any data using external devices.
3. `DADiskUnmountApprovalCallback` – called when a disk is about to be unmounted.

To work with the framework, firstly, a new `DASessionRef` must be created. This is an identifier used by the system to distinguish multiple instances of DA sessions across multiple processes/threads. A new session is created by invoking routine `DASessionCreate(CFAllocatorRef allocator)`. The routine returns `DASessionRef` which should be stored somewhere, where it is accessible to a scope of our application, which is going to be dealing with any parts of DA.

Once the session identifier is obtained, desired callbacks need to be registered. Each callback has its own registration function, which generally corresponds to the name of the callback itself. It takes a dictionary specifying the devices of interest (e.g., unformatted media, USB media, CD media, devices with specified ID, etc.). This parameter can be omitted for working with all devices. Authorization functions have a deadline for the decision — typically a couple of seconds. If they do not return a value in time, the system no longer waits and connects the device.

The description of the disk contains information about its state, such as whether it is formatted, its mount point, bus path, and so on. These information can be obtained with the `DADiskCopyDescription()` function[10], or in rare cases using the `DADiskCopyIOMedia()` call to access the `I/O Registry` object and then retrieve additional data or communicate with the device's interface using `I/O Kit`. Finally, for the code to work, the `Disk Arbitration` session needs to be scheduled in a `RunLoop` or `Grand Central Dispatch (GCD)` queue. A sample code of how to block all mountable devices is attached in the source code of the thesis. The code is based on [16].

Disk Arbitration Internals

`Disk Arbitration` subsystem consists of a daemon (`diskarbitrationd`) and a framework (`DiskArbitration.framework`). The daemon is the central authority for the processing of new device requests and automatic mounting of their partitions. It also notifies clients of disk's attachment and ejection, and fulfills the role of an arbitrator in claiming disks by its clients. More details on this subsystem can be found in [64].

6.7 OpenBSM

`OpenBSM` is a not-well documented subsystem of the `macOS` kernel providing a real-time stream of system information, including file and network operations. `OpenBSM` is an open-source implementation of the Sun BSM (Basic Security Module) security audit API and file format [70]. For Apple, it was implemented by `McAfee` as a part of the effort to get `Common Criteria (CC) Common Access Protection Profile (CAPP)` certification in `Mac OS X 10.3.6 Panther` [1].

Until `Mac OS X 10.5 Leopard`, the subsystem was supported by the kernel, but it had to be installed and configured separately. Later, `OpenBSM` was implemented in `TrustedBSD` based on Darwin source code (where `OpenBSM` part was re-licensed by Apple to `BSD` license upon request). Subsequently, with the support of `Mac OS X 10.6 Snow Leopard`, it became part of the `macOS`[11]. The `TrustedBSD` team also maintained the `macOS` version. The implementation for `TrustedBSD` and `macOS` has been extended to support system-specific features, such as Mach interfaces[12] [37, 49, 64].

[10] A complete list of properties can be found in `DiskArbitration/DADisk.h` header file.
[11] https://ssl.apple.com/support/security/commoncriteria/
[12] http://www.trustedbsd.org/openbsm.html

OpenBSM consists of a set of system calls, libraries for managing audit logs (kernel events, such as system calls, and application events, such as logins and password changes), sample configuration files, and various tools such as `praudit`, or `auditreduce` [70]. The `/etc/security/` folder contains four important files:

- `audit_event` — mapping of events to their identifiers,
- `audit_class` — list of event classes that can be monitored,
- `audit_control` — the global configuration of the subsystem, and
- `audit_user` — local configuration for individual users.

The individual audit logs can be found in `/var/audit` folder. It contains binary files, so its advisable to use, for example, the `praudit -x <file>` command to read records in XML format, or `auditreduce <file>` to filter them before further processing. The default configuration can be seen in Listing 6.4, which logs Login/Logout (lo) and Authorization/Authentication (aa) events into the `/var/audit` folder. Each flag is a group of events and their specification can be found in `/etc/security/audit_event` file. It is also possible to create a custom group. The individual settings are described in more detail by Rich Trouton [69].

```
$ cat /etc/security/audit_control
#
# $P4: //depot/projects/trustedbsd/openbsm/etc/audit_control#8 $
#
dir:/var/audit
flags:lo,aa
minfree:5
naflags:lo,aa
policy:cnt,argv
filesz:2M
expire-after:10M
superuser-set-sflags-mask:has_authenticated,has_console_access
superuser-clear-sflags-mask:has_authenticated,has_console_access
member-set-sflags-mask:
member-clear-sflags-mask:has_authenticated
```

Listing 6.4: Default configuration of the OpenBSM subsystem

One possible disadvantage is that the files generated by `audit(4)` are rotated, so after some time, it may not be possible to discover the past events. Another is the delay to access them (the daemon must first complete them), which brings an inevitable delay in the ability to respond to individual events. This is not suitable for systems that monitor system activity and need to take immediate actions, such as DLP or IDS. A possible solution is a special device `/dev/auditpipe`. Using the device, one can monitor events in real-time. As the `/dev/fsevents` described in section 6.5, it can be cloned, and several clients can subscribe for the events simultaneously. The audit pipe device is blocking by default but supports non-blocking I/O, asynchronous I/O using `SIGIO`, and polling using `select(2)` and `poll(2)`. However, in the case of direct work with the device, unlike `audit(4)` records, we are not guaranteed to receive a record (i.e., if the buffer size set during the cloning is too small, events can be discarded) [2].

```
$ cat /etc/security/audit_class
#
# $P4: //depot/projects/trustedbsd/openbsm/etc/audit_class#6 $
#
0x00000000:no:invalid class
0x00000001:fr:file read
0x00000002:fw:file write
0x00000004:fa:file attribute access
0x00000008:fm:file attribute modify
0x00000010:fc:file create
0x00000020:fd:file delete
0x00000040:cl:file close
0x00000080:pc:process
0x00000100:nt:network
0x00000200:ip:ipc
0x00000400:na:non attributable
0x00000800:ad:administrative
0x00001000:lo:login_logout
0x00002000:aa:authentication and authorization
0x00004000:ap:application
0x10000000:res:reserved for internal use
0x20000000:io:ioctl
0x40000000:ex:exec
0x80000000:ot:miscellaneous
0xffffffff:all:all flags set
```

Listing 6.5: OpenBSM event classes

The `libbsm` library can be used for working with the audit pipe device. It facilitates the reading and writing of data. However, it does not provide the possibility of parsing or filtering records, and for this purpose, it is necessary to process individual (binary) records and tokens manually [63]. Applications can further configure whether to subscribe to events according to global settings or a local set, independent of the global one, using the `ioctl` call [2].

Operations interesting for a DLP solution that can be tracked using `OpenBSM` framework include working with files and processes. A sample code using `OpenBSM` which monitors read operations can be found in the thesis' source files and its output in Listing D.11. The basecode was a project of Alessio Santoru[13]. The main functions of the framework include:

- `au_read_rec()` to read an audit record from the audit pipe file,
- `au_fetch_tok()` to read a token from the audit record, and
- `au_print_flags_tok()` to print the token.

The output of the program can be seen in Listing D.11. In general, the recorded data include `PID`, `PPID`, and network and file operations (including mounts), which, together with timestamps, allow us to create a complete picture of what was happening at a given moment.

It is, again, just a reactive monitoring mode without the possibility to block anything. However, using this framework one can get supplementary information about system events

[13] https://github.com/meliot/filewatcher/

from user-space. For example, `OpenBSM` could be used for maintaining a complete image of running processes in the system, and this information later used for a decision in a blocking part of the program. The `libbsm` library is also used to work with `audit_token` objects used in the `Endpoint Security Framework` described in section 5.3. I need to recall that Patrick Wardle pointed out the inconsistency of the process information which varied based on how the process was created, and it would be convenient to take it into account before working with the framework [72].

OpenBSM Internals Audit support is enabled during system startup based on the `rc.conf(5)` flag, or in the case of `macOS`, `/System/Library/LaunchDaemons/com.apple.auditd.plist`, which is a launchd property list. From the plist one can read, that it is a `auditd(8)` daemon which is responsible for `audit(4)` log files, their rotations, etc. There is also a `audit(8)` utility that can be used to control the daemon. An enhanced version of `praudit(1)` can be found on Jonathan Levin's webpage[14].

6.8 Summary

The section discussed various approaches to monitoring system operations. Several facilities can be used to access system events from user-space, such as `Dtrace`, `kdebug`, and `kevents`. For some, Apple does not provide documentation, and in general, it is a considerable effort even to find them. In addition, their use is rather painful, also due to the need to work with unprocessed raw data. Some of them do not even work in newer versions of the system, due to added system protections, or their use in a production environment is not suitable for possible negative impacts on other system components.

Some frameworks only provide the ability to monitor selected objects, which is suitable for live monitoring, IDS systems, or post mortem analysis, while another provide an interface for authorization of operations like the `Disk Arbitration` framework. The section also described accessing external devices from user-space using the `I/O Kit` API, which can be used to create an application-based driver. Furthermore, the combination of different frameworks can provide the necessary information and the desired effect, although not alone (as exemplified by several blog posts by Patrick Wardle describing the design and functionality of his security tools). Another example is a combination of two frameworks in order to get program arguments of a process [81] (though it was not necessary in this case [53].

`Disk Arbitration` currently provides probably the most suitable proactive solution for blocking external devices. It offers a relatively simple API and supports a wide range of removable media. In addition, it also allows one to mount the device with defined flags, which allows one to intercept the connection process (or to disconnect the currently connected devices) and connect them in read-only mode.

Due to the number of combinations of information that individual frameworks offer, and what options they provide, it depends on the specific use case for choosing the right one. The section did not cover more advanced event tracking techniques (and presumably less supported, or wanted), specifically tracking of Mach messages and hooking/injecting of system calls. More about the tracking of Mach messages can be found in [81].

[14] http://newosxbook.com/tools/supraudit.html

Chapter 7

Current State of External Devices Control

This section discusses current software solutions that are either blocking some type of external devices or implement an approach described in previous chapters to monitor important system events, and are easy to edit in order to provide the desired functionality. Generally, I did not find many products developed enough to be usable. Although I could not find any relevant research papers about the control of external devices, described tools include solutions from companies like Facebook and Google. That is also why this section discusses projects that do not provide needed functionality to achieve expected goals but are in active development, and their functionality may be extended to support required areas.

7.1 Open-source Projects

There are many projects on the Internet using some of the frameworks described in previous chapters, such as `DiskArbitration`, `I/O Kit`, or `KEXT`s. The following is a description of selected projects that are freely available in the form of source code and work with external disks or the `Endpoint Security` framework.

Disk Arbitrator

`Disk Arbitrator` is an application which makes use of `Disk Arbitration` framework in order to block mounting of external devices. The project also supports mounting the devices in read-only mode; however, it does not take into account overwriting the data using direct access to its `/dev` block device by utilities, such as `dd`. The project aims to provide a forensic utility allowing security researchers to control the process of disk mounts, for example, to deny the system to modify its content after it is connected (i.e., because of file system repairs, `fseventsd`, or `spotlight` indexing). Furthermore, it also takes into account different mount parameters when mounting in read-only mode, needed in different scenarios. Although the project does not support devices already attached during its launch, the framework itself supports it, and it is possible to extend the code with this functionality without more significant architectural changes [33].

Sinter

`Sinter` is entirely user-mode client, leveraging the `Endpoint Security` framework. The application can be used to block execution of specified applications (e.g., cloud clients) based on the hash value of their code directory. The functionality of `Sinter` will be extended in the future, but so far, blocking of application executions is not enough to restrict neither disk mounts nor access to data stored on cloud [68].

Santa

`Santa` is a product of Google's Macintosh Operations Team using which one can — similarly to `Sinter` — prevent the execution of unknown binaries. `Santa` code used `Kauth` API to listen for and authorize executions and a user-space daemon for policy decisions. Now it uses the new `Endpoint Security` framework. It is used in enterprises to restrict binaries that can run on the machines. Although `Santa` can log file operations and currently supported functionality may be used to, for example, prevent users from installing cloud clients, it is not enough to achieve the goal of this thesis [43].

Mount Stop Daemon

`Mount Stop Daemon` is a short implementation of a `mountstopd` daemon blocking mounts of newly attached devices. Compared to the `Disk Arbitrator` projects, its implementation is straightforward, and except blocking, it does not provide any additional functionality [77].

Direct IO

`Direct IO` works on `FreeBSD`, `Linux, macOS`, and `Windows` and blocks devices and files by exclusively opening them. As described in the previous chapter, this is not a suitable approach for a DLP solution [62].

OSQuery

`OSQuery` is an open-source framework developed by Facebook used in many companies to monitor their machines. Currently supported operating systems include `Linux`, `macOS`, `Windows`, and `FreeBSD`. `OSQuery` provides a wide range of information about the operating system, such as running processes, information about mounted disks, loaded kernel extensions, hardware events, browser plugins, and more through `SQL` requests. An example of available information one can get about mounted disks can be seen in `device_*` and `disk_*` tables accessible through its documentation[1]. In the part of the source code[2] which is responsible for `macOS` system events can be seen that it uses the system frameworks described in previous chapters, such as `FSEvents API`, `Disk Arbitration`, `I/O Kit`, and `OpenBSM`. Although this project offers a large amount of information about the system, it is only a reactive monitoring approach that is not suitable for a DLP solution. There is also missing information about the application responsible for logged action. More information can be found in the `OSQuery` documentation [39].

[1] https://osquery.io/schema/4.3.0/#device_file
[2] https://github.com/osquery/osquery/tree/master/osquery/events/darwin

Providence

`Providence` is a project written in `Go` which uses the `Endpoint Security` framework to monitor process and file events. It supports `macOS`, `BSD`, and `Linux` but does not seem to provide blocking abilities [52].

USB Detective

`USB Detective` is a swift application watching for connected `HID` devices that may be malicious. It uses a `USBDeviceSwift` library[3], which is a Swift wrapper for working with `I/O Kit` from user-space as was described in chapter 6. Such option is suitable for communication with `I/O Kit` objects from user-space, and although it provides a way to send raw requests to the device's interface, it is not suitable for a DLP solution that needs a proactive way to block or alter device mounts [36].

Little Flocker

`Little Flocker` was a tool developed by Jonathan Zdziarski. It used the `MACF` framework to watch access of applications to files on disk and asked a user for a decision if to allow it or not. It was somewhat equivalent to the `Little Snitch` firewall but watching files. It was an open-source utility; however, after the author joined Apple, the tool was purchased by `F-Secure` and the original application neither its source is available anymore. Forks of the original repository do not exist either [42].

7.2 Commercial Products

There are several security companies offering their products for endpoint security. Most of them offer monitoring of events with the ability to block external devices. Some of them also differentiate cloud services as a part of their DLP functionality, and secure these channels of a possible data leak. Several Endpoint Management products make use of Configuration Profiles[4], which can be used to block specified external devices, mount them in read-only mode, and also restrict iCloud synchronization capabilities. This approach is not discussed in this thesis as it is more of a standardization of endpoint configuration throughout a company, and it is not a suitable way of proactive device control. A DLP solution needs to be flexible to react to system events and take immediate actions. Follows a list of some of the commercial products:

- Hands Off![5],
- CoSoSys Endpoint Protector[6],
- Symantec Data Loss Prevention[7],
- Citrix Endpoint Management[8],

[3] https://github.com/Arti3DPlayer/USBDeviceSwift
[4] https://developer.apple.com/business/documentation/Configuration-Profile-Reference.pdf
[5] https://www.oneperiodic.com/products/handsoff/
[6] https://www.endpointprotector.com/solutions/data-loss-prevention-DLP-for-Mac-OS-X
[7] https://help.symantec.com/home/DLP15.0?locale=EN_US
[8] https://docs.citrix.com/en-us/citrix-endpoint-management

- Code42 Data Loss Protection[9],
- McAfee Total Protection for Data Loss Prevention[10],
- Digital Guardian Endpoint DLP[11],
- BlackBag SoftBlock[12],
- Safetica for macOS[13], and others.

7.3 Summary

There is a lot of commercial products available that have capabilities to block external devices and have control over files being copied to and from cloud storage. Although there are also open-source projects that can block external storage devices and could be relatively easily incorporated in a DLP solution, I did not find any project that would control operations with network drives or a cloud storage. The following chapter discusses a possible approach of implementing a software solution that can control files that are being transferred to and from such locations.

[9] https://www.code42.com/blog/macos-catalina-creates-kernel-crisis-for-legacy-dlp/
[10] https://www.mcafee.com/enterprise/en-us/products/total-protection-for-data-loss-prevention.html
[11] https://digitalguardian.com/products/endpoint-dlp
[12] https://www.blackbagtech.com/products/softblock/
[13] https://www.safetica.com/safetica-for-mac

Chapter 8

Implementation

This chapter describes an implementation of various approaches to control possible channels of data leakage based on the knowledge from previous chapters. Since this work focuses more heavily on the research, and the goal was to find a way how to control access to external storage, and especially cloud drives, it provides a generic prototype which implements the full proposed functionality and could be later incorporated into a more extensive full-fledged software solution. Some of the information the solution is based on have been obtained through undocumented means, it is unfortunately not possible to guarantee that the proposed implementation will continue working even in future releases.

The included proof-of-concept code shows a working approach, and it is possible that when implemented in the production code, its architecture will have to be modified in order to meet Apple requirements, such as placing the `Endpoint Security` code in an application bundle. Furthermore, the final product has to be signed with a valid `SYSX`-enabled certificate in order to run. Table 8.1 contains summary of various frameworks that can be used for monitoring and controlling external disks and file operations. Obviously, all frameworks with blocking abilities can do just audit.

Table 8.1: Frameworks Summary

	Audit	**Control**
External Disks		Disk Arbitration I/O Kit DriverKit
File Operations	Dtrace kdebug kevents FSEvents OpenBSM	Kauth MACF ESF

8.1 USB Device Control

At first, I tried to implement a driver using the `DriverKit` framework described in section 5.2. Based on the work I did with the `I/O Kit` described in section 4.4, I assumed that I might be able to use a similar approach of filtering devices. This way, all USB devices could be blocked and not just the mountable ones, as in the case of `Disk Arbitration` described in section 6.6. As the base of the project, I used Scott Knight's[1] implementation which is based on code snippets shown during the introdution of the framework [23]. I managed to properly set up Xcode so that it built the code without a certificate, and then correctly signed it with the correct entitlements and `Info.plist` files so the system successfully ran it and matched against the desired device. However, a preprocessor macro needed to implement custom DriverKit callbacks could not be used as a code that used it did not compile, even though it was used as in the documentation's example. Because of it, while the driver is loaded, it cannot communicate with the device. If the problem with the macro can be solved, then it might be possible to get a working driver.

Apple had updated the documentation regarding the `DriverKit` SDK couple of weeks before the thesis was submitted, which did not give me enough time to incorporate it into my work, despite my best efforts. Moreover, based on examination of `Info.plist` files of Apple drivers[2], indices in Apple's documentation [26], the fact that there is no `::Probe()` method to be overridden, and error messages I got when I tried to use wildcards instead of device IDs, it seems, that it is not possible to write a generic driver that would match against all USB devices as is possible in the case of the `I/O Kit`. Unfortunately, I was unable to get a working prototype done. The demo code that matches to a USB device but does not communicate with the device can be found in the source files attached with the thesis, and a procedure that has to be followed in order to run it successfully can be found in Listing D.5.

I also considered making a PCI driver or a nub, which would eliminate the need for a separate driver for each type of device. Although I did not have enough time to explore this area, it seems that because of the layered architecture of drivers, it is not easy, if not impossible, to get identifiers of the connected device needed to decide about blocking it. Neither other frameworks that are able to block mount operations (i.e., `Kauth`, `MACF`, `ESF`) provide enough of relevant information about the hardware except the file system type and mount destination. The `Disk Arbitration` framework provides an easy way of blocking external devices and also has an ability to obtain an `I/O Kit` device object. Although it cannot affect the communication of the system with the device, for example, to implement custom device encryption, so far, it is the best way of blocking external devices. Moreover, it can also re-mount devices with custom parameters, for example, in read-only mode. It provides an elegant and relatively simple user-space solution which can also, if necessary, obtain an `I/O Registry` object to get the device interface and subsequently communicate with the device using `I/O Kit`.

As an example of cooperation of two modules — one blocking cloud drives and another one blocking external disks — the final project leveraged the `Disk Arbitration` framework and used it to block all USB drives, regardless of their identifiers. An advantage of using the `Disk Arbitration` framework is that it works in user-space and it does not collide with drivers of a virtualized environment. The code works with local devices, that is, those that have respective `/dev/` device (thus not network mounts).

[1] https://github.com/knightsc/USBApp

[2] For instance `/System/Library/DriverExtensions/com.apple.DriverKit-AppleUSBFTDI.dext`

8.2 Cloud Storage Control

Cloud drives are not treated by the operating system as external devices, but rather synchronized folders of individual cloud client applications. Therefore, it is not possible to block them using drivers or any checks of external file systems, but rather at the level of file operations. Although `ESF` does not support operating systems older than `macOS 10.15 Catalina`, the majority of Apple product owners use the latest versions of their system as can be seen on statistics of our clients or freely available counters on the Internet[3]. Therefore, the `Endpoint Security` Framework — described in section 5.3 — was the best choice for the implementation of file operations control. In general, at the lowest level, I divide file operations into two groups — ones that can modify file or folder content, and others that are just meant to access the content. All operations from the latter group are allowed if the blocking is not in full-blocking mode, and operations that tend to modify files, such as `rename` or `create`, are in both read-only and full-blocking mode denied. An exception is the `open` operation, in which the file opening flags are modified as needed — if in read-only mode, flags allowing modification are removed, otherwise either all flags are cleared or none based on the blocking mode. No blocking applies to white-listed apps in any mode, and neither internal cloud application operations are blocked, which is described further.

As a part of blocking, also access to the root folder of the cloud storage is restricted, so there can be no leak of sensitive information, for example, from the names of files in the folder. In addition to the file operation itself, details of the process that performed the operation are needed as well. Currently, the process is identified based on its bundle ID, but it can be very easily extended to check the signing certificate of the process' binary, or its path. A problem that may occur in the future using the chosen approach is that if a cloud client application uses any external commands (e.g., `mktemp`) it will not be recognized as an action of the cloud application, and it will be blocked. This can be resolved by monitoring process creations and their parents, which can be done using `ESF` as well.

iCloud

iCloud synchronizes contacts, calendars, photos, and documents across Apple devices and keeps them up to date. It also provides a web interface with support for simple operations — folder creation, file upload, file download, and operations rename, move, and remove. iCloud can also be used by installed third-party applications for data storage.

The application data are stored in `~/Library/Mobile Documents/` folder, and a folder that is shown as the iCloud folder by Finder is `~/Library/Mobile Documents/com~apple~CloudDocs`. If Finder has enabled the option to show hidden files, when the iCloud default folder is open, Finder also shows folders of applications as it would be located in the documents folder, though it is not.

Although iCloud's internals are not documented, some information can be obtained from the manual pages. Execution of `apropos Cloud` command reveals several system daemons responsible for data synchronization, such as `cloudd(8)`, `cloudphotod(8)`, `coredatad(8)`, and `bird(8)`. The daemon that is in charge of backing user documents in the iCloud drive is `bird(8)`. As part of iCloud structure, I controlled only synchronized data folder. In fact, data can be leaked through iCloud Photos as much as through any other synchronized data — calendar, keychain objects, browser bookmarks, and so on.

[3] https://gs.statcounter.com/macos-version-market-share/desktop/worldwide

Dropbox

Dropbox provides a web interface and a `macOS` client application. The application uses one folder, and only that one is synchronized. However, the user has an option to change its location. The user also has a possibility of synchronizing some folders from the home folder, namely the `Desktop`, `Downloads`, and `Documents` folders. If the user chooses so, these folders are moved to the shared Dropbox folder, and symbolic links pointing to them are created in the original locations. Dropbox application's data are stored in the `~/.dropbox` folder, which also contains `Info.json` file. This file contains paths to shared folders linked to individual Dropbox accounts. The implemented solution reads this configuration to locate the Dropbox folder.

Shortly before the thesis was submitted, the Dropbox client application had been updated and a custom file manager was added as a part of the application. Unlike iCloud, which uses a separate service to keep data in sync, Dropbox processes data by just one process. This means that when a user modifies a shared file in the Finder, the Dropbox process sends, and updates the file on the servers. If the file is modified on the server, the Dropbox process updates it locally. The same process with the same PID works with files within the provided GUI and also sends them to the server. Therefore it is not easy to distinguish which changes were initiated by the user and which are only replicated from the server.

Local Changes

If a file is modified locally, there is no difference between operations performed by the Finder, Terminal, or Dropbox. All are performed in the same way as in other non-shared folders, so deleting a file moves it into the trash bin and can be restored using Finder (the Dropbox GUI application does not provide a trash bin yet, unlike its web interface). Also adding a new file is done directly in-place.

Remote Changes

If a new file is added at a remote location, the Dropbox application downloads the file into its cache folder located at `<dropbox_folder>/.dropbox.cache/new_files/` and then moves it into the shared folder. When a file is deleted in remote storage, the local file is moved into the cache folder as well, in this case, into the `old_files` folder. Within the Dropbox's GUI app, a user cannot access Dropbox configuration folders, cache including. And it is not because it is a hidden folder, as these are fully accessible and being correctly synced.

In case of modifying a currently existing file, for example, renaming or moving it, the Dropbox process modifies the specific file in-place; thus, there is no easy way to distinguish it from a local change. The solution would be to prohibit the launch or use of the GUI application in other ways and allow all its activity in the background, or to include User Behavior Analysis, including tapping mouse clicks and key presses, and a type of an application in the foreground in combination with Dropbox's network activity. However, it still does not exclude race conditions, which could cause false negatives and subsequent data loss.

Usually, `macOS` applications have settings (such as if it should use Finder or other application for browsing files) stored in `~/Library/Preferences` and data in `~/Library/Application Support` folders. However, it seems that it is not the case of Dropbox. While I was changing Dropbox's settings, the application worked with several files in ~/`.dropbox` folder. Based on the operations it performed, I assume that its settings are stored in an encrypted database ~/`.dropbox/instanceX/config.dbx`, while the instance number is probably stored in — again encrypted — ~/`.dropbox/instance_db/instance.dbx`. However, I judge this only on the basis of the observed behavior of the Dropbox application and I cannot confirm it without the database password.

The project code implements an approach of blocking of some operations of Dropbox application incorrectly to show an example of how to distinguish several operations. It blocks some operations that cannot be decided whose action it is, but Dropbox finishes the synchronization once the blocking is stopped. However, there could be some not–synchronized file conflicts that the Dropbox might not be able to resolve. For real use and proper syncing, it is more convenient to white-list the Dropbox application and to take care of its GUI separately.

8.3 Application Design

The project is implemented as a command-line application in Objective-C++. The source code includes an Xcode project as well as a `Makefile` for manual compilation. In order to successfully run the binary file, it must be signed with a certificate and with correct entitlements. The `sign.sh` file serves for this purpose. It is also necessary to turn off system's SIP protection. During the program run, debug reports are being printed, and their granularity can be set using program's arguments. Before the program terminates, it prints statistics, such as numbers of processed and dropped events. Listing 8.1 contains a sample output during tests were running.

```
20200608010111 [II] ./Clouds/base.mm:131:<HandleEvent>: (RONLY) ES_EVENT_TYPE_NOTIFY_ACCESS - operation at '/Users/jk/tmp/Dropbox' by com.apple.finder(353)
20200608010111 [II] ./Clouds/base.mm:131:<HandleEvent>: (RONLY) ES_EVENT_TYPE_AUTH_READDIR - ALLOWING operation at '/Users/jk/tmp/Dropbox' by com.apple.finder(353)
20200608010112 [II] ./Clouds/base.mm:131:<HandleEvent>: (RONLY) ES_EVENT_TYPE_NOTIFY_CLOSE - operation at '/Users/jk/tmp/Dropbox/very_special_file.txt alias' by com.apple.finder(353)
20200608010112 [II] ./Clouds/base.mm:131:<HandleEvent>: (RONLY) ES_EVENT_TYPE_AUTH_OPEN - ALLOWING (FREAD,FNONBLOCK,O_DIRECTORY), BLOCKING () operation at '/Users/jk/tmp/Dropbox' by com.getdropbox.dropbox(451)
20200608010112 [II] ./Clouds/base.mm:131:<HandleEvent>: (RONLY) ES_EVENT_TYPE_AUTH_OPEN - ALLOWING (FREAD,FNONBLOCK,O_SYMLINK), BLOCKING () operation at '/Users/jk/tmp/Dropbox/blockerd.GpWQG' by com.getdropbox.dropbox(451)
20200608010112 [II] ./Clouds/base.mm:131:<HandleEvent>: (RONLY) ES_EVENT_TYPE_AUTH_UNLINK - BLOCKING operation at '/Users/jk/tmp/Dropbox' '/Users/jk/tmp/Dropbox/blockerd.remove.QnJ00' by com.apple.rm(90451)
20200608010112 [II] ./Clouds/base.mm:131:<HandleEvent>: (RONLY) ES_EVENT_TYPE_AUTH_RENAME - BLOCKING operation at '/Users/jk/tmp/Dropbox/blockerd.rename.u3h0t' '/Users/jk/tmp/Dropbox/blockerd.rename_dst.24df9' by com.apple.mv(90456)
20200608010113 [II] ./Clouds/base.mm:131:<HandleEvent>: (RONLY) ES_EVENT_TYPE_AUTH_CREATE - BLOCKING operation at '/Users/jk/tmp/Dropbox/blockerd.duplicate.2749f' by com.apple.cp(90461)
```

Listing 8.1: Sample output

The most important classes in the source code are `DiskBlocker::` and `CloudBlocker::`. Classes are initialized from the main `Blocker::` class. Both classes are initialized in their `::Init()` method, where the `ESF` client is created and callback functions registered. `DiskBlocker::` registers methods for monitoring of newly added devices, their removal, renaming, and a mount approval callback.

`CloudBlocker::` registers to receive events specified by `m_eventsOfInterest` member variable, and `ESF` calls one code-block for all events. Because calling of blocking operations directly from the block handler in combination with a high rate of events cloud cause the framework's buffer to fill with events, and as a result, the `ESF` client would be killed [82], the individual events are copied, set for processing asynchronously, and the handler returns almost immediately. Events are processed in `CloudBlocker::HandleEvent()` method which calls another method asynchronously — `CloudBlocker::HandleEventImpl()`. That is because the program has to respond before the deadline specified by the `ESF`, and thus the first thread must not block. If the second thread does not return before the deadline, the first one responds in favor of the application that requested the operation.

8.4 Summary

The initial requirements and use-cases heavily influence the choice of the correct approach for our solution. For example, an application working with file events might need to use a combination of frameworks to get all the information it needs. Some solutions may require access to the raw data which is worked on, whether it is a case of external, network, or cloud storage, while others not. The chapters discussed a general approach on how to restrict access to external disks and cloud storage. Reasons for deciding whether to block a given operation or not can be relatively easily extended by information from other code modules, which, for example, scan files on the disk for sensitive content.

Some clouds use a separate process for synchronization and some take care of everything in a single process. Currently, the most suitable implementation for the supported cloud providers is to control file operations using the `Endpoint Security` framework. However, it may not be the case for every cloud provider, and previous chapters should help to decide on the best approach, whether it is blocking of processes, disks, or files.

Chapter 9

Testing

Because of the need to have `System Integrity Protection` disabled in order to run the project, the application was tested in a virtual machine running `macOS 10.15.4 Catalina`, which could affect overall system performance. The following chapters analyze the impact of the application on system's performance and validity of the implementation. The implementation of the blocking depends on the internals of every cloud provider, thus each of them was tested separately.

9.1 Performance Tests

Although the goal of the implementation was to show a principle of the blocking, and not to be as effective as possible, performance tests were done to have an idea about the system slowdown. The results can be seen in Figure 9.1 and tables shown in Appendix B. The graph represents times that are in the tables and shows a difference between two sets of three tests. In the first run, various batches of files were copied, and the time of the operation was measured. During the second run, the same files were copied but with a blocker running in a read-only mode. In this run, there were no files blocked because none of them destinated in a cloud folder. Though each of them had been checked by the blocker. Finally, in the last test, the same amount of files was copied into a cloud folder, thus all of them were supposed to be blocked and none should end up in the destination folder. The read-only mode was selected because of additional processing needed compared to full-blocking mode, for example, examination and modification of `open` operation flags. All three runs were timed and logged. These tests timed copying of up to one-thousand files and each set of tests was run with 1kB files and 1MB files. The test with larger files was added to lower the rate of new events in case the program could not handle smaller files, to reveal an edge of how many operations the implementation can process. But as can be seen in the results, it was able to process all the files without any errors.

As can be seen in Figure 9.1, both runs where all the operations were blocked took approximately the same time. This makes sense, as no files were effectively copied so it shows how fast the events could be processed. I tried to run tests with the `test` command to keep the time needed for processing files by the utility being measured as short as possible, but it generated just access check notifications, which cannot be blocked. Each test was timed five times and the average value was drawn in the graph. During all the tests, all events were processed with no drops or errors, as can be seen in the tables of Appendix B. Numbers in the columns „Allowed“ and „Denied“ mean the number of copied files, not a number of allowed and denied operations, respectively. In fact, the application processed around 3-6x more events than shown in the table. This was because every copied file generated several events, and some events were generated by the operating system with tests themselves. That is why I did not show a combination of „Allowed“ and „Denied“ operations as, technically, it was the case of all the tests. Scripts used during the testing, as well as the results are attached with the thesis sources.

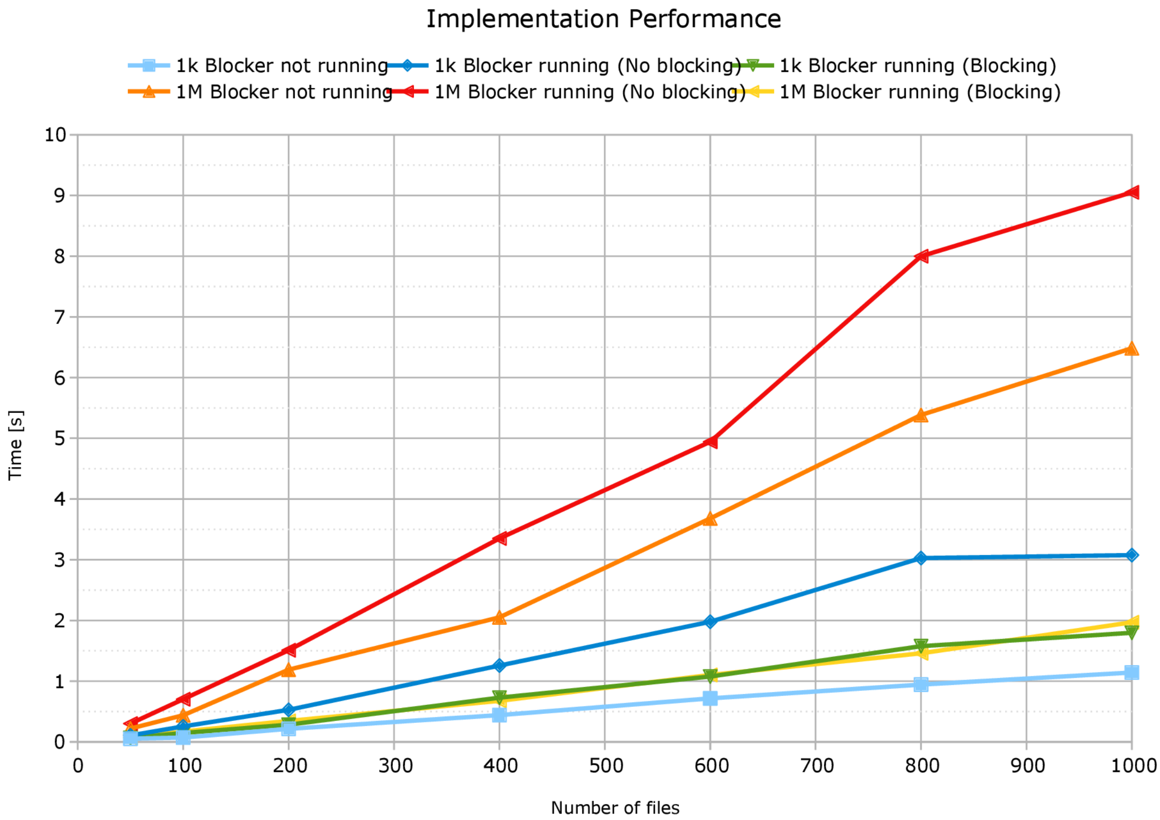

Figure 9.1: Implementation Performance

9.2 Validity of the Implementation

The implementation validity check consisted of performing operations that were to be allowed or denied and checks on whether or not they were performed. Tests of command-line utilities were performed using an automated script, and testing of GUI applications was done manually. The script is attached with the thesis sources.

The results can be seen in the tables of Appendix B. `Copy` operations mean copying files out of the cloud location, and `Clone` operations represent copying within the cloud folder. The difference is that the former one is expected to be allowed in the read-only mode. Normally, Dropbox's GUI does not provide a direct way to enter a folder outside the Dropbox's shared folder; however, it is can be achieved by placing a symbolic link in the Dropbox's folder pointing to the desired location.

Even when the solution runs in the full-blocking mode, Dropbox keeps displaying a structure of the shared folder. However, it displays just cached content, and since it cannot open the real folder, it can also show currently non-existent files or, conversely, not contain those that have been added and the application had been blocked before they could be indexed. In this case, opening existing files still fails because of blocking, even if the user sees it in the cache. In the case of tests of synchronization with a remote device (in this case, a web interface), operations marked as „denied“ were denied only locally.

Furthermore, Dropbox's synchronization does not work in the full-blocking mode at all because the application tries to open the shared folder at first. If it fails (due to blocking), it does not try to download the remote file into its cache folder. Because of this, no remote operation was replicated locally. The goal is to keep all background operations needed for proper synchronization allowed, and block other user and system processes (i.e., spotlight indexing) to prevent unwanted disclosure of information. Unfortunately, as was described in the previous chapter, in some cases, there is not an easy way to distinguish between user's and synchronization operations.

On the other hand, iCloud uses a separate program for data synchronization; therefore, it is very effortless to distinguish user operations from the synchronization. Unlike Dropbox, when deleting a remote file, it does not move it into a recycle bin in the cloud, but in the device itself. Therefore, recovery of the file while blocking is enabled is possible only on another device that does not have blocking enabled.

9.3 Summary

The chapter discussed the results of the implementation tests. As can be seen in the graph and tables, the interception of file operations causes a certain overhead, but the implementation was not focused on speed as much as to show work with the framework for cloud–control purposes, and it could be indeed optimized.

Tests showed that in the case of iCloud, its background functionality was preserved, and the data were correctly synchronized while the user had restricted access as desired. On the contrary, in the case of Dropbox, the implementation was complicated by the fact that Dropbox performed all file operations under one process — whether locally or remotely — and thus complicated the blocking decisions.

During speed tests, the program managed to process all data, without any dropped events or failed file operations. At the same time, it was able to decide which operations should be allowed and which should not.

Chapter 10

Conclusion

There are many possible channels of data leakage in the operating system. The thesis focuses on external storage, specifically USB disks and cloud drives. The thesis was written in such a way that a reader could understand it without previous experience with `macOS` kernel development, and leverage frameworks described in the thesis.

Firstly, the thesis introduced a reader history of Apple operating systems and described its architecture. Then it covered various techniques that can be used for system audit and control. `macOS` provides several ways of accessing external devices. Some of them are in a form of the kernel extension, such as drivers, and some can be done from the user-space. However, for blocking devices, there are just two convenient approaches — drivers and the `DiskArbitration` framework. Both frameworks used for driver development require the developer to be assigned a special entitlement in order to deploy the drivers. Otherwise, the system protections must be disabled in order to load the drivers, which is not recommended even on the development machine. Furthermore, loading of drivers is more complicated during development in a virtualized environment. In case private KPIs are used — which have hitherto been necessary to get various information and capabilities needed for security software to work — there is a risk of change of their interfaces and binary incompatibility between major OS releases, and their usage is not recommended. So far, the most suitable option seems to be the Disk Arbitration framework, both for easy implementation, backward compatibility, and also future support.

Due to the nature of the project and the possible change in the behavior of cloud applications that synchronize user data, the thesis discusses the possibilities of monitoring and controlling the system in a broader manner, so that the acquired knowledge can be applied to other cloud providers that would not work as currently supported ones. The individual frameworks were not easy to find and explore because, in some cases, it is not an Apple-supported approach or even if supported — it is entirely undocumented, and in these cases, the most relevant sources of information were reports by security engineering reversing their functionality or describing CVEs. System Extensions are quite a new technology, thus all book sources and most of the internet sources have led to old approaches. Throughout the work, I kept discovering new frameworks and possibilities of system monitoring/control but the topic is too extensive to be covered in this thesis.

Cloud drives are not being processed by the system as external drives, so the thesis covered `KEXT`s to control file operations as well. During work on the thesis, a new version of the operating system was released, so the content of the thesis was expanded to include also newly supported System Extensions, which were eventually used. The implementation works in user-space and uses the `Endpoint Security` Framework. Furthermore, how different cloud disks work internally was described, and also a way of blocking specific clouds while preserving their synchronization capabilities. In the case of iCloud, its full background functionality was preserved; however, blocking of Dropbox with keeping Dropbox's file manager enabled was not so successful.

The implemented result is an open-source solution that can control work with selected cloud drives and blocks attachment of all USB drives. Its advantage is that it works in user-space, thus it can be relatively easily incorporated into a larger DLP software and possibly extended by more advanced decision-making methods. In the future, I would like to optimize the implementation to minimize the impact on the system's performance, explore other cloud providers, and develop a driver using `DriverKit` to explore its capabilities further.

Bibliography

[1] *AUDIT(4) BSD Kernel Interfaces Manual.* March 2009.

[2] *AUDITPIPE(4) BSD Kernel Interfaces Manual.* October 2010.

[3] Apple Computer Company. *Apple Basic Users Manual* [article]. Palo Alto, CA: Apple Computer Company, october 1976 [cit. 2019-10-24]. Available at: https://www.applefritter.com/files/basicman.pdf.

[4] Apple Inc.. *Deprecated Kernel Extensions and System Extension Alternatives - Support - Apple Developer* [online]. [cit. 2020-02-12]. Available at: https://web.archive.org/web/20200212095834/https://developer.apple.com/support/kernel-extensions/.

[5] Apple Inc.. *File System Events | Apple Developer Documentation* [online]. [cit. 2020-05-30]. Available at: https://web.archive.org/web/20200530142632/https://developer.apple.com/documentation/coreservices/file_system_events?language=objc.

[6] Apple Inc.. *Inside Mac OS X - Kernel Environment.* November 2000.

[7] Apple Inc.. *Mac OS X: Kernel — Session 106.* 2000. Apple Worldwide Developers Conference.

[8] Apple Inc.. *Leveraging BSD Services in Mac OS X — Session 139.* 2003. Apple Worldwide Developers Conference.

[9] Apple Inc.. *Introduction to Accessing Hardware From Applications* [online]. February 2007 [cit. 2020-06-08]. Available at: http://web.archive.org/web/20200608194751/https://developer.apple.com/library/archive/documentation/DeviceDrivers/Conceptual/AccessingHardware/AH_Intro/AH_Intro.html.

[10] Apple Inc.. *Technical Q&A QA1574: Kernel's MAC framework* [online]. January 2008 [cit. 2020-05-29]. Available at: https://web.archive.org/web/20200529155600/https://developer.apple.com/library/archive/qa/qa1574/_index.html.

[11] Apple Inc.. *Kernel Extension Programming Topics* [online]. 2010 [cit. 2020-05-27]. Available at: https://web.archive.org/web/20200529115203/https://developer.apple.com/library/archive/documentation/Darwin/Conceptual/KEXTConcept/KEXTConceptIntro/introduction.html.

[12] Apple Inc.. *Technical Note TN2127: Kernel Authorization* [online]. March 2010 [cit. 2020-05-29]. Available at: https://web.archive.org/web/20200529135718/https://developer.apple.com/library/archive/technotes/tn2127/_index.html.

[13] Apple Inc.. *Technical Q&A QA1319: Installing an I/O Kit KEXT Without Rebooting* [online]. 2011 [cit. 2020-05-28]. Available at: https://web.archive.org/web/20200529115231/https://developer.apple.com/library/archive/qa/qa1319/_index.html.

[14] Apple Inc.. *File System Events Programming Guide* [online]. 2012 [cit. 2020-05-30]. Available at: https://web.archive.org/web/20200529233338/https://developer.apple.com/library/archive/documentation/Darwin/Conceptual/FSEvents_ProgGuide/Introduction/Introduction.html.

[15] Apple Inc.. *Network Kernel Extensions Programming Guide* [online]. 2012 [cit. 2020-05-29]. Available at: https://web.archive.org/web/20200529115103/https://developer.apple.com/library/archive/documentation/Darwin/Conceptual/NKEConceptual/intro/intro.html.

[16] Apple Inc.. *Disk Arbitration Programming Guide* [online]. 2013 [cit. 2020-01-20]. Available at: https://web.archive.org/web/20200120051547/https://developer.apple.com/library/archive/documentation/DriversKernelHardware/Conceptual/DiskArbitrationProgGuide/Introduction/Introduction.html.

[17] Apple Inc.. *Kernel Programming Guide* [online]. 2013 [cit. 2020-01-18]. Available at: https://web.archive.org/web/20200118045607/https://developer.apple.com/library/archive/documentation/Darwin/Conceptual/KernelProgramming/Architecture/Architecture.html.

[18] Apple Inc.. *Introduction to I/O Kit Fundamentals* [online]. April 2014 [cit. 2020-06-08]. Available at: http://web.archive.org/web/20200608194822/https://developer.apple.com/library/archive/documentation/DeviceDrivers/Conceptual/IOKitFundamentals/Introduction/Introduction.html.

[19] Apple Inc.. *KauthORama* [online]. 2014 [cit. 2020-06-06]. Available at: https://web.archive.org/web/20200606073227/https://developer.apple.com/library/archive/samplecode/KauthORama/Introduction/Intro.html.

[20] Apple Inc.. *About Developing for Mac* [online]. 2015 [cit. 2020-01-18]. Available at: https://web.archive.org/web/20200118022742/https://developer.apple.com/library/archive/documentation/MacOSX/Conceptual/OSX_Technology_Overview/About/About.html%23//apple_ref/doc/uid/TP40001067.

[21] Apple Inc.. *MacOS Security – Overview for IT* [online]. 2018 [cit. 2020-01-18]. Available at: https://www.apple.com/ca/business/docs/resources/macOS_Security_Overview.pdf.

[22] Apple Inc.. *Mac_policy.h* [online]. 2019 [cit. 2020-05-07]. Available at: https://web.archive.org/web/20200529112842/https://opensource.apple.com/source/xnu/xnu-6153.61.1/security/mac_policy.h.auto.html.

[23] Apple Inc.. *System Extensions and DriverKit — Session 702.* 2019. Apple Worldwide Developers Conference.

[24] Apple Inc.. *Action_type - es_message_t | Apple Developer Documentation* [online]. 2020 [cit. 2020-04-08]. Available at: https://web.archive.org/web/20200529115005/https://developer.apple.com/documentation/endpointsecurity/es_message_t/3228968-action_type?language=objc.

[25] Apple Inc.. *Client | Apple Developer Documentation* [online]. 2020 [cit. 2020-04-08]. Available at: https://web.archive.org/web/20200408191300/https://developer.apple.com/documentation/endpointsecurity/client?language=objc.

[26] Apple Inc.. *Creating a Driver Using the DriverKit SDK | Apple Developer Documentation* [online]. 2020 [cit. 2020-06-08]. Available at: http://web.archive.org/web/20200608084236/https://developer.apple.com/documentation/driverkit/creating_a_driver_using_the_driverkit_sdk?language=objc.

[27] Apple Inc.. *Deadline - es_message_t | Apple Developer Documentation* [online]. 2020 [cit. 2020-04-08]. Available at: https://web.archive.org/web/20200529113421/https://developer.apple.com/documentation/endpointsecurity/es_message_t/3334985-deadline?language=objc.

[28] Apple Inc.. *Endpoint Security | Apple Developer Documentation* [online]. 2020 [cit. 2020-04-08]. Available at: https://web.archive.org/web/20200408191453/https://developer.apple.com/documentation/endpointsecurity?language=objc.

[29] Apple Inc.. *Es_event_type_t - EndpointSecurity | Apple Developer Documentation* [online]. 2020 [cit. 2020-04-14]. Available at: https://web.archive.org/web/20200529113506/https://developer.apple.com/documentation/endpointsecurity/es_event_type_t?language=objc.

[30] Apple Inc.. *Installing System Extensions and Drivers | Apple Developer Documentation* [online]. 2020 [cit. 2020-04-13]. Available at: https://web.archive.org/web/20200529115037/https://developer.apple.com/documentation/systemextensions/installing_system_extensions_and_drivers?language=objc.

[31] Apple Inc.. *Message | Apple Developer Documentation* [online]. 2020 [cit. 2020-04-08]. Available at: https://web.archive.org/web/20200529113342/https://developer.apple.com/documentation/endpointsecurity/message?language=objc.

[32] Apple Inc.. *NetworkExtension | Apple Developer Documentation* [online]. 2020 [cit. 2020-04-23]. Available at: https://web.archive.org/web/20200529113313/https://developer.apple.com/documentation/networkextension?language=objc.

[33] Burghardt, A. and Wojniak, K. *Disk-Arbitrator.* GitHub, 2020. Available at: https://github.com/aburgh/Disk-Arbitrator.

[34] Chisnall, D. *What Is Mac OS X* [online]. February 2010 [cit. 2020-01-19]. Available at: https://web.archive.org/web/20200118230423/https://www.informit.com/articles/article.aspx?p=1552774.

[35] Dalrymple, M. and Hillegass, A. *Advanced Mac OS X Programming: The Big Nerd Ranch Guide*. Big Nerd Ranch, 2011. Big Nerd Ranch guides. ISBN 9780321706256.

[36] Dutchcoders. *Usb Detective*. GitHub, 2019. Available at: https://github.com/dutchcoders/usbdetective.

[37] EDGE, C. and O'Donnell, D. *Enterprise Mac Security: Mac OS X*. Apress, 2015. ISBN 9781484217122.

[38] Efrat, E. *Recent Security Enhancements in NetBSD* [online]. September 2006 [cit. 2020-05-29]. Available at: www.netbsd.org/~elad/recent/recent06.pdf.

[39] Facebook. *Welcome to osquery - osquery* [online]. [cit. 2020-05-26]. Available at: https://osquery.readthedocs.io/en/stable/.

[40] fG!. *Abusing OS X TrustedBSD framework to install r00t backdoors...* [online]. September 2011 [cit. 2020-05-29]. Available at: https://web.archive.org/web/20200529155213/https://reverse.put.as/2011/09/18/abusing-os-x-trustedbsd-framework-to-install-r00t-backdoors/.

[41] fG!. *Can I SUID: a TrustedBSD policy module to control suid binaries execution* [online]. October 2014 [cit. 2020-05-29]. Available at: https://web.archive.org/web/20200529160041/https://reverse.put.as/2014/10/03/can-i-suid-a-trustedbsd-policy-module-to-control-suid-binaries-execution/.

[42] Fleishman, G. *Little Flocker reincarnates as Xfence, a free beta from F-Secure | Macworld* [online]. April 2017 [cit. 2020-06-03]. Available at: https://web.archive.org/web/20200603205834/https://www.macworld.com/article/3190149/little-flocker-reincarnates-at-f-secure-in-free-beta.html.

[43] Google. *Santa*. GitHub, 2020. Available at: https://github.com/google/santa.

[44] Gross, S. *Don't Trust the PID!* 2018. WarCon IT Security Conference.

[45] Knight, S. *Virus scanning on macOS | Scott Knight* [online]. October 2018 [cit. 2020-05-29]. Available at: https://web.archive.org/web/20200529160333/https://knight.sc/reverse%20engineering/2018/10/19/virus-scanning-on-macos.html.

[46] Knight, S. *CVE-2019-8805 - A macOS Catalina privilege escalation | Scott Knight* [online]. October 2019 [cit. 2020-04-21]. Available at: https://web.archive.org/web/20200529113138/https://knight.sc/reverse%20engineering/2019/10/31/macos-catalina-privilege-escalation.html.

[47] Knight, S. *System Extension internals | Scott Knight* [online]. August 2019 [cit. 2020-04-21]. Available at: https://web.archive.org/web/20200529113205/https://knight.sc/reverse%20engineering/2019/08/24/system-extension-internals.html.

[48] Kotyk, A. *Avoid Kernel Development in macOS with System Extensions and DriverKit.* [online], 02. April 2020 [cit. 2020-04-12]. Available at: https://web.archive.org/web/20200529113237/https://www.apriorit.com/dev-blog/669-mac-system-extensions.

[49] Levin, J. *Mac OS X and iOS Internals: To the Apple's Core*. 1st ed. Wrox, 2012. ISBN 978-1118057650.

[50] Levin, J. *The Apple Sandbox: Deeper Into The Quagmire*. 2016. Hack In he Box Security Conference.

[51] Levin, J. *Endpoint Security* [online]. 2019 [cit. 2020-04-21]. Available at: https://web.archive.org/web/20200529113047/http://newosxbook.com/articles/eps.html.

[52] livingstonetech. *Providence*. GitHub, 2020. Available at: https://github.com/livingstonetech/providence.

[53] Lu, K. *Monitoring macOS, Part I: Monitoring Process Execution via MACF* [online]. March 2018 [cit. 2020-05-29]. Available at: https://web.archive.org/web/20200529162934/https://www.fortinet.com/blog/threat-research/monitoring-macos--part-i--monitoring-process-execution-via-macf.html.

[54] Lu, K. *Monitoring macOS, Part II: Monitoring File System Events and Dylib Loading via MACF* [online]. March 2018 [cit. 2020-05-29]. Available at: https://web.archive.org/web/20200529163317/https://www.fortinet.com/blog/threat-research/monitor-file-system-events-and-dylib-loading-via-macf-on-macos.html.

[55] Lu, K. *Monitoring macOS, Part III: Monitoring Network Activities Using Socket Filters* [online]. March 2018 [cit. 2020-05-29]. Available at: https://web.archive.org/web/20200529112557/https://www.fortinet.com/blog/threat-research/monitoring-macos--part-iii--monitoring-network-activities-using-.html.

[56] Miller, C. and Zovi, D. *The Mac Hacker's Handbook*. Wiley, 2011. ISBN 9781118080337.

[57] Oakley, H. *Watching macOS file systems: FSEvents and volume journals - The Eclectic Light Company* [online]. September 2017 [cit. 2020-05-30]. Available at: https://web.archive.org/web/20200530150856/https://eclecticlight.co/2017/09/12/watching-macos-file-systems-fsevents-and-volume-journals/.

[58] Oakley, H. *Booting the Mac: Visual Summary - The Eclectic Light Company* [online]. August 2018 [cit. 2020-05-26]. Available at: https://web.archive.org/web/20200529112728/https://eclecticlight.co/2018/08/25/booting-the-mac-visual-summary/.

[59] Oakley, H. *Why Catalina may have problems with extensions - The Eclectic Light Company* [online]. October 2019 [cit. 2020-01-20]. Available at: https://web.archive.org/web/20200105121706/https://eclecticlight.co/2019/10/21/why-catalina-may-have-problems-with-extensions/.

[60] Oracle. *DTrace User Guide* [online]. 2010 [cit. 2020-05-29]. Available at: https://web.archive.org/web/20200529224049/https://docs.oracle.com/cd/E19253-01/819-5488/.

[61] Perla, E. and Oldani, M. *A Guide to Kernel Exploitation: Attacking the Core*. 1st ed. Elsevier, 2010. ISBN 978-1597494878.

[62] Ronomon. *Direct IO*. GitHub, 2019. Available at: https://github.com/ronomon/direct-io.

[63] Santoru, A. *Real-time auditing on macOS with OpenBSM - meliot's blog* [online]. July 2017 [cit. 2020-06-03]. Available at: https://web.archive.org/web/20200603065000/https://meliot.me/2017/07/02/mac-os-real-time-auditing/.

[64] Singh, A. *Mac OS X Internals: A Systems Approach*. 1st ed. Addison-Wesley Professional, 2006. ISBN 978-0321278548.

[65] Singh, A. *A Technical History of Apple's Operating Systems* [online]. 2006 [cit. 2020-01-17]. Available at: https://web.archive.org/web/20200117163518/http://osxbook.com/book/bonus/chapter1/.

[66] Siracusa, J. *Mac OS X 10.5 Leopard: the Ars Technica review | Ars Technica* [online]. October 2007 [cit. 2020-05-30]. Available at: https://web.archive.org/web/20200530155034/https://arstechnica.com/gadgets/2007/10/mac-os-x-10-5/7/.

[67] Stavonin, A. *Working with TrustedBSD in Mac OS X | System Development* [online]. July 2013 [cit. 2020-05-29]. Available at: https://web.archive.org/web/20200529153356/https://sysdev.me/trusted-bsd-in-osx/.

[68] Trail Of Bits. *Sinter*. GitHub, 2020. Available at: https://github.com/trailofbits/sinter.

[69] Trouton, R. *OpenBSM auditing on Mac OS X | Der Flounder* [online]. January 2012 [cit. 2020-06-03]. Available at: https://web.archive.org/web/20200603001454/https://derflounder.wordpress.com/2012/01/30/openbsm-auditing-on-mac-os-x/.

[70] TrustedBSD. *TrustedBSD - OpenBSM* [online]. [cit. 2020-06-03]. Available at: https://web.archive.org/web/20200603000546/http://www.trustedbsd.org/openbsm.html.

[71] Usov, A. *MacOS security – Secure Kernel Extension Loading (SKEL) - MacAdmins IL* [online]. January 2019 [cit. 2020-01-20]. Available at: https://web.archive.org/web/20200120025700/https://macadmins.co.il/2019/01/21/macos-security-secure-kernel-extension-loading-skel/.

[72] Wardle, P. *Monitoring Process Creation via the Kernel (Part I) | Objective-See's Blog* [online]. November 2015 [cit. 2020-05-29]. Available at: https://web.archive.org/web/20200529160344/https://objective-see.com/blog.html#blogEntry9.

[73] Wardle, P. *Monitoring Process Creation via the Kernel (Part II) | Objective-See* [online]. November 2015 [cit. 2020-05-29]. Available at: https://web.archive.org/web/20200603142140/https://objective-see.com/blog/blog_0x0A.html.

[74] Wardle, P. *Towards Generic Ransomware Detection | Objective-See's Blog* [online]. April 2016 [cit. 2020-05-30]. Available at: https://web.archive.org/web/20200530154140/https://objective-see.com/blog/blog_0x0F.html.

[75] Wardle, P. *Writing a File Monitor with Apple's Endpoint Security Framework | Objective-See's Blog* [online]. September 2019 [cit. 2020-04-21]. Available at: https://web.archive.org/web/20200529112952/https://objective-see.com/blog/blog_0x48.html.

[76] Wardle, P. *Writing a Process Monitor with Apple's Endpoint Security Framework | Objective-See's Blog* [online]. September 2019 [cit. 2020-04-21]. Available at: https://web.archive.org/web/20200529113020/https://objective-see.com/blog/blog_0x47.html.

[77] Waterhouse, G. *Mount Stop Daemon*. GitHub, 2012. Available at: https://gist.github.com/Calrion/1446123.

[78] Watson, R., Feldman, B., Migus, A. and Vance, C. Design and implementation of the Trusted BSD MAC framework. In:. May 2003, vol. 1, p. 38–49. DOI: 10.1109/DISCEX.2003.1194871.

[79] Watson, R. N. *New approaches to operating system security extensibility*. University of Cambridge, Computer Laboratory, April 2012. 184 p.

[80] Yu, W. *Mac system extensions for threat detection: Part 1 | Elastic Blog* [online]. January 2020 [cit. 2020-04-22]. Available at: https://web.archive.org/web/20200529112923/https://www.elastic.co/blog/mac-system-extensions-for-threat-detection-part-1.

[81] Yu, W. *Mac system extensions for threat detection: Part 2 | Elastic Blog* [online]. January 2020 [cit. 2020-06-03]. Available at: https://web.archive.org/web/20200603075210/https://www.elastic.co/blog/mac-system-extensions-for-threat-detection-part-2.

[82] Yu, W. *Mac system extensions for threat detection: Part 3 | Elastic Blog* [online]. February 2020 [cit. 2020-04-22]. Available at: https://web.archive.org/web/20200529112633/https://www.elastic.co/blog/mac-system-extensions-for-threat-detection-part-3.

[83] Zheng, M., Bai, X. and Qu, H. *All Your Apple Are Belong To Us: Unique Identification and Cross-device Tracking of Apple Devices*. 2019. Black Hat USA.

Appendix A

CD Content

Enclosed CD contains an archive with the|following files and directories:

- `latex/` – $\LaTeX$source files.
- `code/` – The implementation source code.
- `xzuzel00-DP.pdf` – PDF version of the Master's thesis.
- `README` – Text file containing instructions how to compile the tool.

Appendix B

Test Results

Table B.1: Validity of the iCloud read-only restrictions

Cloud Provider	Application	Operation	Result
iCloud (Read-Only)	Finder	Open	Read-Only
		Create	Denied
		Copy	Allowed
		Move	Denied
		Remove	Denied
		Rename	Denied
		Clone	Denied
	Terminal	Open	Read-Only
		Create	Denied
		Copy	Allowed
		Move	Denied
		Remove	Denied
		Rename	Denied
		Clone	Denied
		Hardlink	Denied
		Symlink	Allowed
		Symlink (read)	Read-Only
		Symlink (write)	Denied
		Exchange Data	
	Remote Device	Create	Synced
		Rename	Synced
		Remove	Synced
		Move	Synced

Table B.2: Validity of the iCloud full restrictions

Cloud Provider	Application	Operation	Result
iCloud (No-Access)	Finder	Open	Denied
		Create	Denied
		Copy	Denied
		Move	Denied
		Remove	Denied
		Rename	Denied
		Clone	Denied
	Terminal	Open	Denied
		Create	Denied
		Copy	Denied
		Move	Denied
		Remove	Denied
		Rename	Denied
		Clone	Denied
		Hardlink	Denied
		Symlink	Allowed
		Symlink (read)	Denied
		Symlink (write)	Denied
		Exchange Data	
	Remote Device	Create	Synced
		Rename	Synced
		Remove	Synced
		Move	Synced

Table B.3: Validity of the Dropbox read-only restrictions

Cloud Provider	Application	Operation	Result
Dropbox (Read-Only)	Dropbox GUI	Open	Read-Only
		Create	Denied
		Copy	Allowed
		Move	Denied
		Remove	Denied
		Rename	Denied
		Duplicate	Denied
	Finder	Open	Read-Only
		Create	Denied
		Copy	Allowed
		Move	Denied
		Remove	Denied
		Rename	Denied
		Duplicate	Denied
	Terminal	Open	Read-Only
		Create	Denied
		Copy	Allowed
		Move	Denied
		Remove	Denied
		Rename	Denied
		Duplicate	Denied
		Hardlink	Denied
		Symlink	Allowed
		Symlink (read)	Allowed
		Symlink (write)	Denied
		Exchange Data	
	Remote Device	Create	Synced
		Rename	Denied
		Remove	Synced
		Move	Denied
		Copy	Synced
		Restore	Synced

Table B.4: Validity of the Dropbox full restrictions

Cloud Provider	**Application**	**Operation**	**Result**
Dropbox (No-Access)	Dropbox GUI	Open	Denied
		Create	Denied
		Copy	Denied
		Move	Denied
		Remove	Denied
		Rename	Denied
		Duplicate	Denied
	Finder	Open	Denied
		Create	Denied
		Copy	Denied
		Move	Denied
		Remove	Denied
		Rename	Denied
		Duplicate	Denied
	Terminal	Open	Denied
		Create	Denied
		Copy	Denied
		Move	Denied
		Remove	Denied
		Rename	Denied
		Duplicate	Denied
		Hardlink	Denied
		Symlink	Allowed
		Symlink (read)	Denied
		Symlink (write)	Denied
		Exchange Data	
	Remote Device	Create	Denied
		Rename	Denied
		Remove	Denied
		Move	Denied
		Copy	Denied
		Restore	Denied

Table B.5: Implementation performance

File Size	Expected Result			Real Result			Errors			
	Allowed	Denied	Time [s]	Allowed	Denied	Time [s]	Drops	Deadline	Alloc	Reply
1kB	50	0	0,0524	50	0	0,1066	0	0	0	0
	0	50	0,0524	0	50	0,0662	0	0	0	0
	100	0	0,0716	100	0	0,255	0	0	0	0
	0	100	0,0716	0	100	0,1454	0	0	0	0
	200	0	0,2152	200	0	0,528	0	0	0	0
	0	200	0,2152	0	200	0,283	0	0	0	0
	400	0	0,4413	400	0	1,2574	0	0	0	0
	0	400	0,4413	0	400	0,7286	0	0	0	0
	600	0	0,7163	600	0	1,978	0	0	0	0
	0	600	0,7163	0	600	1,0764	0	0	0	0
	800	0	0,9434	800	0	3,0272	0	0	0	0
	0	800	0,9434	0	800	1,5764	0	0	0	0
	1000	0	1,1424	1000	0	3,0762	0	0	0	0
	0	1000	1,1424	0	1000	1,7968	0	0	0	0
1MB	50	0	0,2205	50	0	0,3008	0	0	0	0
	0	50	0,2205	0	50	0,072	0	0	0	0
	100	0	0,438	100	0	0,7028	0	0	0	0
	0	100	0,438	0	100	0,1706	0	0	0	0
	200	0	1,1895	200	0	1,5088	0	0	0	0
	0	200	1,1895	0	200	0,3452	0	0	0	0
	400	0	2,0513	400	0	3,351	0	0	0	0
	0	400	2,0513	0	400	0,676	0	0	0	0
	600	0	3,6798	600	0	4,9426	0	0	0	0
	0	600	3,6798	0	600	1,1028	0	0	0	0
	800	0	5,3843	800	0	7,9994	0	0	0	0
	0	800	5,3843	0	800	1,4584	0	0	0	0
	1000	0	6,4846	1000	0	9,0512	0	0	0	0
	0	1000	6,4846	0	1000	1,97	0	0	0	0

Appendix C

Additional Figures

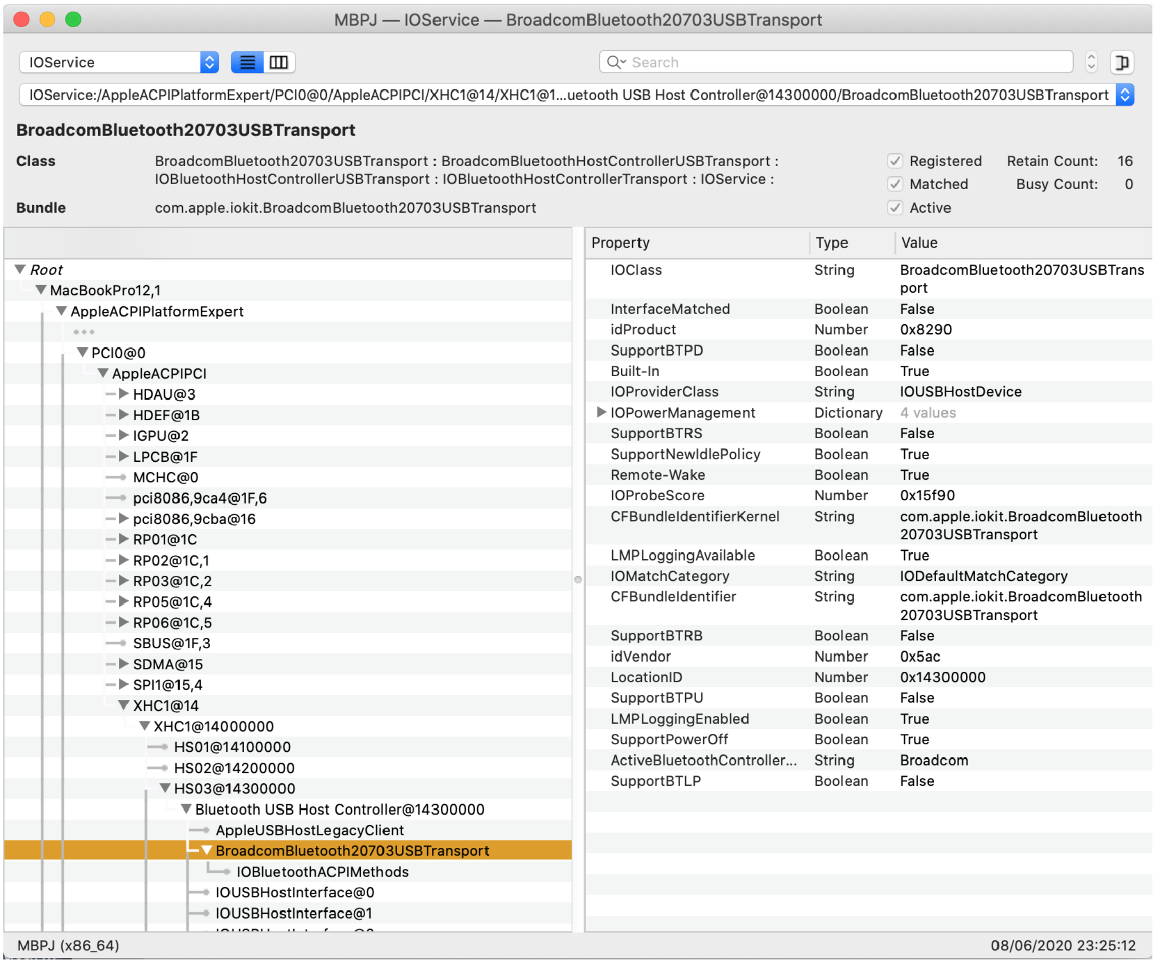

Figure C.1: I/O Registry Explorer

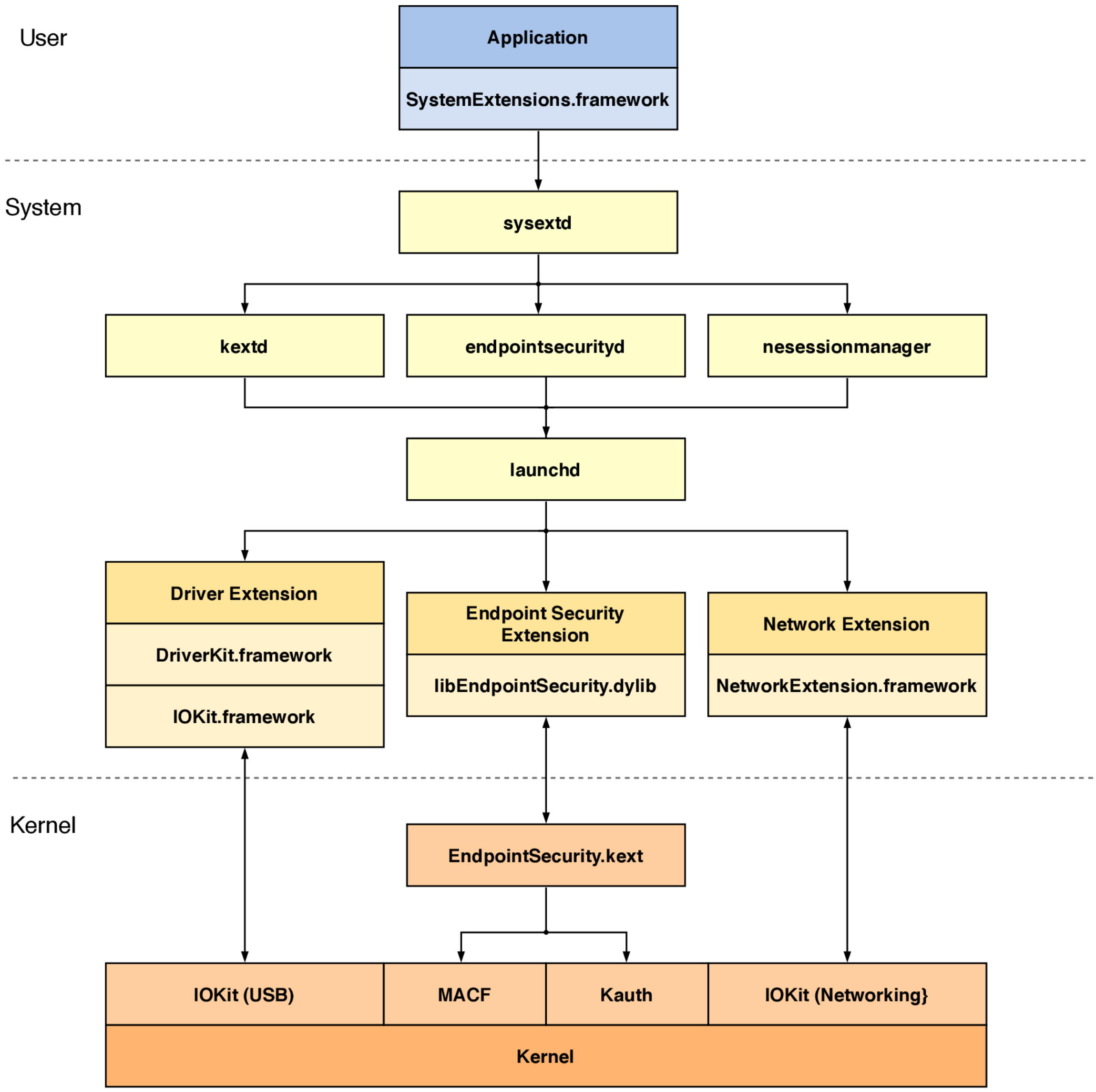

Figure C.2: System extensions subsystem architecture [47]

Appendix D

Sample Outputs

```c
#include <sys/systm.h>
#include <mach/mach_types.h>

static const char* g_demoName = "generic-kext";
// Define endpoints
kern_return_t generic_kext_demo_start(kmod_info_t * ki, void *d);
kern_return_t generic_kext_demo_stop(kmod_info_t *ki, void *d);

// The extension has been loaded. Register your callbacks..
kern_return_t generic_kext_demo_start(kmod_info_t * ki, void *d)
{
    printf("(%s) Hello, World!\n", g_demoName);
    return KERN_SUCCESS;
}

// Clean up allocated resources.
kern_return_t generic_kext_demo_stop(kmod_info_t *ki, void *d)
{
    printf("(%s) Goodbye, World!\n", g_demoName);
    return KERN_SUCCESS;
}
```

Listing D.1: Generic KEXT demo

```
# Check libraries it needs to be linked against
jz@macos-10 demos % kextlibs -xml -undef-symbols /tmp/generic-kext\ demo.kext
<key>OSBundleLibraries</key>
    <dict>
        <key>com.apple.kpi.libkern</key>
        <string>19.4</string>
    </dict>
# Build with updated Info.plist
# Set proper permissions
jz@macos-10 demos % sudo cp -R DerivedData/IDP-dema/Build/Products/Debug/generic-kext\ demo.kext /tmp/
# Check the driver is ready for loading
jz@macos-10 demos % sudo kextutil -nt /tmp/generic-kext\ demo.kext
Kext with invalid signature (-67050) allowed: <OSKext 0x7ff2e8d26270 [0x7fff8b9d48c0]> { URL = "file:///private/tmp/generic-kext%20demo.kext/", ID = "com.jzlka.generic-kext-demo" }
Code Signing Failure: code signature is invalid
/private/tmp/generic-kext demo.kext appears to be loadable (including linkage for on-disk libraries).
# Load the extension
jz@macos-10 demos % sudo kextload /tmp/generic-kext\ demo.kext
# Check loaded extensions
jz@macos-10 demos % kextstat | grep generic
  141    0 0xffffff7f82e8b000 0x2000     0x2000.    com.jzlka.generic-kext-demo (1) 9062F540-7804-3EB4-B872-39F4079C43A1 <5>
# Unload the extension
jz@macos-10 demos % sudo kextunload /tmp/generic-kext\ demo.kext
# Check that debug messages were printed
jz@macos-10 demos % sudo dmesg | grep generic
(generic-kext) Hello, World!
(generic-kext) Goodbye, World!
```

Listing D.2: Loading of KEXT

```
# Check libraries it needs to be linked against
jz@macos-10 demos % kextlibs -xml -undef-symbols /tmp/kauth\ demo.kext
    <key>OSBundleLibraries</key>
    <dict>
        <key>com.apple.driver.AppleACPIPlatform</key>
        <string>6.1</string>
        <key>com.apple.kpi.bsd</key>
        <string>19.4</string>
        <key>com.apple.kpi.libkern</key>
        <string>19.4</string>
    </dict>
# Build with updated Info.plist
# Set proper permissions
jz@macos-10 demos % sudo cp -R ./DerivedData/IDP-dema/Build/Products/Debug/kauth\
    demo.kext /tmp
# Check the extension is ready for loading
jz@macos-10 demos % sudo kextutil -nt /tmp/IOKit-driver\ demo.kext
Kext with invalid signature (-67050) allowed: <OSKext 0x7f9c93c0c080 [0x7fff8f94b8c0]> {
    URL = "file:///private/tmp/kauth%20demo.kext/", ID = "com.jzlka.kauth-demo" }
Code Signing Failure: code signature is invalid
/private/tmp/kauth demo.kext appears to be loadable (including linkage for on-disk
    libraries).
# Load the extension
jz@macos-10 demos % sudo kextload /tmp/kauth\ demo.kext
# Check loaded extensions
jz@macos-10 demos % sudo kextstat | grep kauth
  142    0 0xffffff7f82ea8000 0x2000     0x2000     com.jzlka.kauth-demo (1)
    56CF3095-3490-3CF9-8557-2B7D86785E52 <5 1>
# Do stuff in the monitored folder
# Unload the extension
jz@macos-10 demos % sudo kextunload /tmp/kauth\ demo.kext
# Check if debug messages were printed
jz@macos-10 demos % sudo dmesg | grep kauth
(kauth)_start: Hello Cruel World!
Point of interest: /tmp/kauth-demo
kauth scope=com.apple.kauth.vnode, action=SEARCH, uid=501, vp=/private/tmp/kauth-demo,
    dvp=<null>
kauth scope=com.apple.kauth.vnode, action=READ_ATTRIBUTES|READ_SECURITY, uid=501,
    vp=/private/tmp/kauth-demo, dvp=<null>
kauth scope=com.apple.kauth.vnode, action=LIST_DIRECTORY, uid=501,
    vp=/private/tmp/kauth-demo, dvp=<null>
kauth scope=com.apple.kauth.vnode, action=LIST_DIRECTORY|SEARCH, uid=501,
    vp=/private/tmp/kauth-demo, dvp=<null>
kauth scope=com.apple.kauth.vnode, action=WRITE_DATA, uid=501,
    vp=/private/tmp/kauth-demo/test.c, dvp=<null>
(kauth)_stop: Goodbye Cruel World!
```

Listing D.3: Loading of Kauth KEXT with sample log

```
# Check libraries it needs to be linked against
jz@macos-10 demos % kextlibs -xml -undef-symbols /tmp/IOKit-driver\ demo.kext
    <key>OSBundleLibraries</key>
    <dict>
        <key>com.apple.kpi.iokit</key>
        <string>19.4</string>
        <key>com.apple.kpi.libkern</key>
        <string>19.4</string>
    </dict>
# Build with updated Info.plist
# Install the extension with proper permissions
jz@macos-10 demos % sudo cp -R ./DerivedData/IDP-dema/Build/Products/Debug/IOKit-driver\ demo.kext /tmp/
# Check the driver is ready for loading
jz@macos-10 demos % sudo kextutil -nt /tmp/IOKit-driver\ demo.kext
Notice: /private/tmp/IOKit-driver demo.kext has debug properties set.
Kext with invalid signature (-67050) allowed: <OSKext 0x7f850440f860 [0x7fff8b9d48c0]> { URL = "file:///private/tmp/IOKit-driver%20demo.kext/", ID = "com.jzlka.IOKit-driver-demo" }
Code Signing Failure: code signature is invalid
/private/tmp/IOKit-driver demo.kext appears to be loadable (including linkage for on-disk libraries).
# Load the extension
jz@macos-10 demos % sudo kextload /tmp/IOKit-driver\ demo.kext
# Check loaded extensions
jz@macos-10 demos % sudo kextstat | grep IOKit
  158    0 0xffffff7f82e8b000 0x2000     0x2000     com.jzlka.IOKit-driver-demo (1) F13A690F-E214-3660-B32D-7327609AAEA1 <5 3>
# Attach a USB
# Unload the extension
jz@macos-10 demos % sudo kextunload /tmp/IOKit-driver\ demo.kext
# Check if debug messages were printed
jz@macos-10 demos % sudo dmesg | grep IOKit
(IOKit-driver) Hello, World!
IOKit-driver demo: Initializing
IOKit-driver demo: Probing...
IOKit-driver demo: Blocking with score 2147483647...
IOKit-driver demo: Starting
(IOKit-driver) Goodbye, World!
IOKit-driver demo: Stopping
IOKit-driver demo: Freeing
```

Listing D.4: Loading of I/O Kit driver with sample log

```
jz@macos-10 tmp % git clone https://github.com/knightsc/USBApp/
# Disable SIP
# Delete entitlements file from the project settings
jz@macos-10 tmp % sed -i '' 's/CODE_SIGN_ENTITLEMENTS = .*/CODE_SIGN_ENTITLEMENTS = "";/'
    USBApp/USBApp.xcodeproj/project.pbxproj
# Fix the development team in the build settings and change the certificate
# Change Derived Data location to project-relative in the project settings
# Build the project
# Sign the bundle and the sysx from terminal
jz@macos-10 tmp % codesign --force -vvvv --entitlements USBApp/USBApp/USBApp.entitlements
    -s - USBApp.app
jz@macos-10 tmp % codesign --force -vvvv --entitlements
    USBApp/MyUserUSBInterfaceDriver/MyUserUSBInterfaceDriver.entitlements -s -
    USBApp.app/Contents/Library/SystemExtensions/sc.knight.MyUserUSBInterfaceDriver.dext

## Useful commands
# Test a signature
codesign -vvv --deep --strict /path/to/binary/or/bundle
# Check signature of installer package
pkgutil --check-signature /path/to/file.pkg
# Check if the software will run with system policies currently in effect
spctl -vvv --assess --type exec /path/to/application
# Secure timestamp
codesign -dvv /path/to/binary/or/bundle
# Check entitlement file validity
plutil -lint <Project_Name.entitlements>
# Verify an app has properly xml entitlement
codesign -d --entitlements :- <path to signed .app or command-line tool>
```

Listing D.5: Building a DEXT

```
(ESF) Hello, World!
Point of interest: /tmp/ESF-demo

*** Occurence found: /private/tmp/ESF-demo
    BLOCKING: ES_EVENT_TYPE_AUTH_CREATE at 105486573905591 of mach time.
--- EVENT MESSAGE ----
event_type: ES_EVENT_TYPE_AUTH_CREATE (44)
version: 2
time: 1591721901.628853033
mach_time: 105486573905591
deadline: 105546573910148
deadline interval: 60000004557 (60 seconds)
action_type: Auth
- process -
  proc.pid: 90799
  proc.ppid: 68624
  proc.original_ppid: 68624
  proc.ruid: 501
  proc.euid: 501
  proc.rgid: 20
  proc.egid: 20
  proc.group_id: 90799
  proc.session_id: 68623
  proc.codesigning_flags: CS_VALID,CS_RESTRICT,CS_DYLD_PLATFORM,CS_SIGNED (0x22000801)
  proc.is_platform_binary: 1
  proc.is_es_client: 0
  proc.signing_id: com.apple.mkdir
  proc.team_id: (null)
  proc.cdhash: 0xed9e4e025e709df6447544c603f867db3e0cb376
  proc.executable.path:
  file.path: /bin/mkdir
  file.path_truncated: 0
  file.stats: TO BE DONE
- event -
sequence number: 10
event.create.destination_type: ES_DESTINATION_TYPE_NEW_PATH
event.create.destination.new_path.dir:
  file.path: /private/tmp
  file.path_truncated: 0
  file.stats: TO BE DONE
event.create.destination.new_path.filename: ESF-demo
```

Listing D.6: Sample ESF output

```
(kdebug) Hello, World!
Point of interest: All the events!
1420373830926234: cpu 0 DBG_TRACE code 0x2 thread 0x9be3f4 DBG_FUNC_END
1420373830927233: cpu 0 DBG_TRACE code 0x3 thread 0x9be3f4 DBG_FUNC_START
1420373830927776: cpu 0 DBG_TRACE code 0x1 thread 0x9be3f4 DBG_FUNC_START
1420373830927809: cpu 0 DBG_TRACE code 0 thread 0x9be3f4
1420373830927848: cpu 0 DBG_TRACE code 0 thread 0x9be3f4
1420373830927886: cpu 0 DBG_TRACE code 0 thread 0x9be3f4
1420373830927924: cpu 0 DBG_TRACE code 0x2 thread 0x9be3f4 DBG_FUNC_END
1420373830928921: cpu 0 DBG_TRACE code 0x3 thread 0x9be3f4 DBG_FUNC_START
```

Listing D.7: Sample kdebug output

```
(kevent) Hello, World!
Point of interest: /tmp/kevent-demo
Event 4 occurred. Filter -4, flags 33, filter flags NOTE_WRITE, filter data 0, path
    /tmp/kevent-demo
Event 4 occurred. Filter -4, flags 33, filter flags NOTE_RENAME, filter data 0, path
    /tmp/kevent-demo
Event 4 occurred. Filter -4, flags 33, filter flags NOTE_WRITE, filter data 0, path
    /tmp/kevent-demo
Event 4 occurred. Filter -4, flags 33, filter flags NOTE_RENAME, filter data 0, path
    /tmp/kevent-demo
Event 4 occurred. Filter -4, flags 33, filter flags NOTE_WRITE|NOTE_LINK, filter data 0,
    path /tmp/kevent-demo
```

Listing D.8: Sample kevent output

```
(FSEvents-API) Hello, World!
Path of interest: /tmp/FSEvents-API-demo

Change 12446910 in /tmp/FSEvents-API-demo/testdir/234, flags
    87040:{kFSEventStreamEventFlagItemInodeMetaMod,kFSEventStreamEventFlagItemModified,
           kFSEventStreamEventFlagItemChangeOwner,kFSEventStreamEventFlagItemIsFile,}
Change 12447044 in /tmp/FSEvents-API-demo/testdir/.234.swp, flags
    70144:{kFSEventStreamEventFlagItemRemoved,kFSEventStreamEventFlagItemModified,
           kFSEventStreamEventFlagItemIsFile,}
Change 12450648 in /tmp/FSEvents-API-demo/testdir, flags
    133120:{kFSEventStreamEventFlagItemRenamed,kFSEventStreamEventFlagItemIsDir,}
Change 12450649 in /tmp/FSEvents-API-demo/testdir2, flags
    133120:{kFSEventStreamEventFlagItemRenamed,kFSEventStreamEventFlagItemIsDir,}
```

Listing D.9: Sample FSEvents (API) output

```
----0/706 bytes.
#Event
    type = FSE_CREATE_FILE
    pid = 129 (logd)
    #Details
    Type                Length  Data
    FSE_ARG_STRING      88      string =
    /var/db/uuidtext/B6/CFDCC4F3A9397487EBB0B440D867D3.T741oh7F
    FSE_ARG_DEV         4       fsid = 80007
    FSE_ARG_INO         8       ino = 1125942856515584
    FSE_ARG_MODE        4       mode = ?-----x--- (40008, vnode type ?
    FSE_ARG_UID         4       uid = 262155(?)
    FSE_ARG_GID         4       unknown
    FSE_ARG_DONE        45887
----158/706 bytes.
#Event
    type = FSE_CHOWN
    pid = 129 (logd)
    #Details
    Type                Length  Data
    FSE_ARG_STRING      88      string =
    /var/db/uuidtext/B6/CFDCC4F3A9397487EBB0B440D867D3.T741oh7F
    FSE_ARG_DEV         4       dev = 80007(major 0 minor 524295)
    FSE_ARG_INO         8       ino = 1125942856515584
    FSE_ARG_MODE        4       mode = ?-----x--- (40008, vnode type ?)
    FSE_ARG_UID         4       uid = 262155(?)
    FSE_ARG_GID         4       unknown
    FSE_ARG_DONE        45887
----316/706 bytes.
#Event
    type = FSE_RENAME
    pid = 129 (logd)
    #Details
    Type                Length  Data
    FSE_ARG_STRING      88      string =
    /var/db/uuidtext/B6/CFDCC4F3A9397487EBB0B440D867D3.T741oh7F
    FSE_ARG_DEV         4       dev = 80007(major 0 minor 524295)
    FSE_ARG_INO         8       ino = 1125942856515584
    FSE_ARG_MODE        4       mode = ?-----x--- (40008, vnode type ?)
    FSE_ARG_UID         4       uid = 262155(?)
    FSE_ARG_GID         4       unknown
    FSE_ARG_STRING      79      string =
    /var/db/uuidtext/B6/CFDCC4F3A9397487EBB0B440D867D3
    FSE_ARG_DONE        45887
----557/706 bytes.
...
```

Listing D.10: Sample FSEvents (fseventsd) output

```
<record version="11" event="open(2) - read" modifier="0" time="Wed May 13 14:41:58 2020"
        msec=" + 275 msec" >
    <argument arg-num="2" value="0x0" desc="flags" />
    <path>/etc/master.passwd</path>
    <path>/private/etc/master.passwd</path>
    <attribute mode="100600" uid="root" gid="wheel" fsid="16777220" nodeid="1889397"
               device="0" />
    <subject audit-uid="ja" uid="root" gid="wheel" ruid="root" rgid="wheel" pid="95147"
             sid="100006" tid="50331650 0.0.0.0" />
    <return errval="success" retval="6" />
    <identity signer-type="0" signing-id="OpenBSM demo" signing-id-truncated="no"
              team-id="VN555WY3S4" team-id-truncated="no"
              cdhash="0xdf26cadb85b3091c4ac77a95a60373774ac290f6" />
</record>

<record version="11" event="close(2)" modifier="0" time="Wed May 13 14:41:58 2020"
        msec=" + 275 msec" >
    <argument arg-num="2" value="0x3" desc="fd" />
    <path>/Volumes/Data/Ine/Konfiguracne/dotfiles/.gitconfig</path>
    <attribute mode="100700" uid="ja" gid="staff" fsid="16777225" nodeid="95592"
               device="0" />
    <subject audit-uid="ja" uid="ja" gid="staff" ruid="ja" rgid="staff" pid="95155"
             sid="100006" tid="50331650 0.0.0.0" />
    <return errval="success" retval="0" />
    <identity signer-type="1" signing-id="com.apple.git" signing-id-truncated="no"
              team-id="" team-id-truncated="no"
              cdhash="0x86a6f1f54fbeb65b4cf678c1eabfce473c912228" />
</record>
```


Listing D.11: Sample OpenBSM output