TECHNISCHE UNIVERSITÄT DARMSTADT

ANALYSING THE macOS BLUETOOTH STACK

DAVIDE TOLDO

Bachelor Thesis

December 23, 2019

Secure Mobile Networking Lab
Department of Computer Science

SECURE MOBILE NETWORKING

Analysing the macOS Bluetooth Stack
Bachelor Thesis
SEEMOO-BSC-0167

Submitted by Davide Toldo
Date of submission: December 23, 2019

Advisor: Prof. Dr.-Ing. Matthias Hollick
Supervisor: Jiska Classen

Technische Universitaet Darmstadt
Department of Computer Science
Secure Mobile Networking Lab

ABSTRACT

Since Bluetooth is one of the most widely used wireless technologies, it is an interesting subject for security researchers and hackers. Especially in the Apple ecosystem, Bluetooth and Bluetooth Low Energy (BLE) are used for many features, to the point where devices might send public BLE advertisements continuously.

While the Linux Bluetooth stack is open-source and some tools for security research do exist, on macOS it is not even possible to send arbitrary commands to the Host-Controller Interface (HCI) Controller of the Bluetooth chip.

In this thesis, we reverse engineer the Bluetooth stack and building upon the open-source platform *InternalBlue*, a macOS port is implemented, enabling full HCI access, monitoring of packets and connection to *Wireshark*. Additionally to the HCI communication, also the transmission of Asynchronous Connection-Less (ACL) packets is reversed and implemented in a proof of concept.

Using *InternalBlue*, it is possible to read, write and execute the Bluetooth chip's memory on any MacBook, iMac, Mac Mini and Mac Pro, amongst lots of other features that are part of *InternalBlue*—all of which is usually completely inaccessible from within macOS. Furthermore, it is analysed which Mac computers use which Bluetooth chips as well as which firmware versions they have. This helps identify models with specific security flaws or missing features.

ZUSAMMENFASSUNG

Da Bluetooth eine der am weitesten verbreiteten drahtlosen Technologien ist, ist es ein interessantes Thema für Sicherheitsforscher und Hacker. Insbesondere im Apple-Ökosystem werden Bluetooth und BLE für viele Funktionen verwendet, bis hin zu dem Punkt, dass diese Geräte für manche Funktionen kontinuierlich öffentliche BLE Pakete senden.

Während der Bluetooth-Stack unter Linux Open-Source ist und einige Tools für die Sicherheitsforschung existieren, ist es unter macOS nicht einmal möglich, beliebige Befehle an den HCI Controller des Bluetooth-Chips zu senden.

In dieser Arbeit werden wird der Bluetooth-Stack reverse-engineered und aufbauend auf der Open-Source-Plattform *InternalBlue* ein macOS-Port implementiert, der den vollen HCI-Zugriff, die Überwachung von Paketen und die Verbindung zu *Wireshark* ermöglicht. Zusätzlich zur HCI-Kommunikation wird auch die Übertragung von ACL-Paketen reversed und in einem Proof of Concept implementiert.

Mit *InternalBlue* ist es möglich, den Speicher des Bluetooth-Chips auf jedem MacBook, iMac, Mac Mini und Mac Pro zu lesen, zu schreiben und auszuführen, neben vielen anderen Funktionen, die Teil von *InternalBlue* sind - all dies ist normalerweise unter MacOS komplett unzugänglich. Darüber hinaus wird analysiert, welche Mac Computer welche Bluetooth-Chips verwenden und welche Firmware-Versionen haben. Dies hilft, Modelle mit bestimmten Sicherheitslücken oder fehlenden Funktionen zu identifizieren.

ACKNOWLEDGMENTS

I would like to express my deepest gratitude to my parents and my family for supporting me in all the years of my studies and also while writing this thesis.

Special thanks for giving helpful advice while writing this thesis goes to Prof. Dr.-Ing. Matthias Hollick and Jiska Classen.

Furthermore, I especially thank Jiska Classen and Luca Toldo for proof-reading my thesis.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LISTINGS

ACRONYMS

ACL Asynchronous Connection-Less
ADB Android Debug Bridge
API Application Programming Interface
AWDL Apple Wireless Direct Link

BLE Bluetooth Low Energy
BSD Berkeley Software Distribution

FIFO First-In-First-Out

GCD Grand Central Dispatch
GUI Graphical User Interface

HCI Host-Controller Interface

IOBE IOBluetoothExtended

Kexts Kernel Extensions

L2CAP	Logical Link Control and Adaption Protocol
LKM	Loadable Kernel Modules
LMP	Link Manager Protocol
MAC	Medium Access Control
MBP	MacBook Pro
OS	Operating System
OWL	Open Wireless Link
PCIe	Peripheral Component Interconnect Express
RAM	Random Access Memory
ROM	Read-Only Memory
SIP	System Integrity Protection
TCP	Transmission Control Protocol
UART	Universal Asynchronous Receiver-Transmitter
UDP	User Datagram Protocol
UI	User Interface
USB	Universal Serial Bus
XNU	X is Not Unix
XPC	Cross-Process Communication

1 INTRODUCTION

While macOS is one of the most widely used operating systems on the world, its Bluetooth stack and its inner workings have been very obscure and unknown to date. The investigation of the macOS Bluetooth stack is especially interesting, since Apple uses Bluetooth for lots of their services and features, like Continuity and *AirDrop*. The Apple Watch also uses Bluetooth and Bluetooth Low Energy (BLE) for all data transfer which includes medical data.

Apple provides high-level frameworks for developers, but they don't give us all the freedom and possibilities that would hardware-wise be possible. It is also very hard to research security topics relating to Bluetooth without having direct access to the chip.

In this thesis, we uncover the veil and document how Bluetooth works in macOS.

We gain full access to the Broadcom Bluetooth chip and show all of the quirks and challenges on the way there. An implementation in the form of a macOS port for *InternalBlue* is included. Both the transmission of Host-Controller Interface (HCI) packets (between host and Bluetooth chip) and Asynchronous Connection-Less (ACL) packets (between host and connected devices) are reverse engineered and we show how to access these features, deep in the Bluetooth stack.

1.1 MOTIVATION

Research tools that allow full access to off-the-shelf devices' Bluetooth chips in macOS are very limited. In systems like Linux and Android, the Bluetooth chip's HCI controller offers local sockets that a process can connect to. Commands and resulting events are exchanged using basic socket communication. In macOS, there are frameworks that are supposed to be used but offer limited functionality, which makes them useless for security research. This is due to them being designed specifically for Apps that typically don't need more than the functionality provided by `CoreBluetooth` and the publicly available `IOBluetooth` and `IOBluetoothUI` functions. Apple's own tools and utilities in macOS do however have full access, so using their own methods via `IOBluetooth`'s private API we can gain the same amount of control, too.

1.2 CONTRIBUTIONS

BLUETOOTH STACK DOCUMENTATION This thesis provides detailed documentation of the Bluetooth stack in macOS and all the various levels that Bluetooth communication typically traverses. Reverse engineering of a private API is shown as well as an introduction to Kernel extensions and low-level device communication in macOS.

INTERNALBLUE PORT A full-featured macOS port of *InternalBlue* [7] is implemented that offers full HCI access, memory dumping, writing to memory and executing custom assembly on the Bluetooth chip amongst many other features that allow doing Bluetooth security research on macOS.

CROSS-LANGUAGE HCI FRAMEWORK The simple socket-based architecture of our Objective-C framework enables Bluetooth security research on macOS from different programming languages.

REVERSE ENGINEERING ACL Almost all data transfer, including music streaming uses ACL, a protocol that has also been analysed in this thesis. We show how it works in macOS and how to use it directly, bypassing the official Apple APIs for full access.

MACOS BUG We document a bug in macOS that allows to read system memory and potentially use the Bluetooth memory as a side-channel.

MAC BLUETOOTH HARDWARE ANALYSIS We discover the Bluetooth hardware strategy that Apple applies for the Mac and identify obsolete firmware versions on new devices.

1.3 OUTLINE

Firstly, we provide background information on the macOS Bluetooth stack architecture, introduce Kernel extensions and explain concurrency in macOS in Chapter 2.

In Chapter 3, related work that helped to understand the architecture is presented as well as work that pokes in other Apple Bluetooth stacks.

Chapter 4 then explains in more detail how exactly the task of gaining full HCI and ACL access on macOS was accomplished and how the low-level code is loaded into the Python project *InternalBlue*.

A performance analysis about how `DispatchQueues` affect the speed of Bluetooth memory dumps as well as an overview of Mac models with their Bluetooth chips and firmware versions constitute Chapter 5.

Chapter 6 covers various alternative approaches for the implementation of *InternalBlue* on macOS and why they were not chosen for the final version.

Finally, Chapter 7 wraps up this thesis with our conclusion and some final thoughts.

2 BACKGROUND

This chapter provides background information about Bluetooth, Kernel extensions and concurrency in macOS. Reverse engineering various frameworks and Kernel Extensions allowed us to document the Bluetooth stack in macOS (Section 2.1). In Section 2.2 we explain how Kernel Extensions work and lastly talk about concurrency using `DispatchQueues`, an important concept when communicating asynchronously between chips in Section 5.1.

2.1 MACOS BLUETOOTH STACK

XPC is Apple's low-level interprocess communication mechanism.

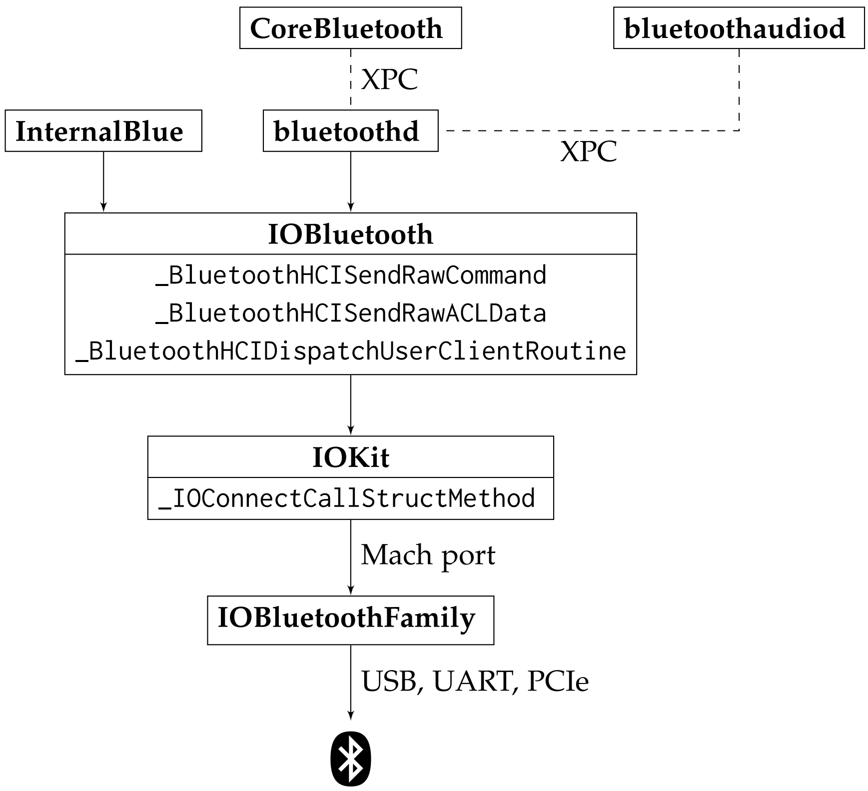

Figure 2.1: Overview of the macOS Bluetooth stack.

The macOS Bluetooth stack works very differently to other platforms like Linux and even iOS. While in Linux you can open Transmission Control Protocol (TCP) sockets on the loopback interface to communicate with the Host-Controller Interface (HCI) Controller on the Bluetooth chip, this is not possible in macOS. It is supposed to be a

Similarly to IOBluetoothFamily.kext for Bluetooth, IO80211Family.kext is the main 802.11 Wi-Fi driver.

security feature and all communication is required to go through the `IOBluetoothFamily.kext` driver.

A hack is needed to send arbitrary commands to the chip via the `IOBluetooth` framework's private Application Programming Interface (API), something usually not possible when using the official method through `CoreBluetooth.framework` and the public `IOBluetooth.framework` functions.

2.1.1 *Bluetooth communication in macOS*

In macOS, Kernel Extensions (Kexts) play a very important role. Like the name implies, they extend Apple's X is Not Unix (XNU) kernel by low-level (usually driver) functionality, but are not compiled into the kernel and instead loaded at boot time or via the Berkeley Software Distribution (BSD) `kextload` utility.

A Kext can be a device driver, add support for another filesystem or provide new system calls.

The Kernel Extension `IOBluetoothFamily.kext` is essentially a driver that communicates directly with the Bluetooth chip and is the lowest layer of the Bluetooth stack. It receives requests from `IOKit` via Mach messages and sends them to the Bluetooth chip over Universal Serial Bus (USB), Universal Asynchronous Receiver-Transmitter (UART) or Peripheral Component Interconnect Express (PCIe).

Usually, when developing Mac applications, you are supposed to use the `CoreBluetooth` or public `IOBluetooth` API inside of your Objective-C or Swift code. `CoreBluetooth` passes calls via Cross-Process Communication (XPC) messages through a `CBXpcConnection` to the `com.apple.bluetoothd` Bluetooth daemon. `bluetoothd` issues a function call to `IOBluetooth`'s `IOBluetoothHostController` object, which then uses the `IOKit` Framework to pass commands to the driver.

CoreBluetooth, IOBluetooth and IOKit frameworks can be found in /System/Library/Frameworks.

It is usually only possible to use high-level methods of the documented, public `IOBluetoothHostController` API that is part of `CoreBluetooth`. Some of these methods do for example allow to retrieve the local hostname and Medium Access Control (MAC) address, but not much more than that. Sending arbitrary HCI commands is not officially supported or documented.

Another interesting finding was that the Bluetooth audio daemon `bluetoothaudiod` registers as a *"Bluetooth daemon client"* (`BTDClient`) to communicate with `bluetoothd` over XPC.

2.1.2 *Data Flow Example*

Using the example of a connection setup, we show the whole data flow from the highest layer framework down to the device driver.

COREBLUETOOTH Firstly, `CoreBluetooth.framework` is imported into a Swift or Objective-C project and the `CBCentralManager`'s function `void -[CBCentralManager connectPeripheral:options:]` is called.

This function uses `void -[CBXpcConnection sendMsg:args:]` to send the message that it wants to connect to a peripheral over XPC to the Bluetooth daemon.

BLUETOOTHD In the next step, `bluetoothd` conforms to the `NSXPCListenerDelegate` and declares `DaemonNSXPCClient` as its `exportedObject` which handles XPC messages. `-[CBXPCManager handleCoreBluetoothCommand:]` receives the message from the Bluetooth daemon. Afterwards, in this case, the command is handled by `-[CachedBluetoothDevice performSDPQuery:uuids:]`, which then uses `IOBluetooth`'s `-[IOBluetoothDevice openConnection:]` to open a new connection to the target device. This function produces log output in the terminal as shown in Listing 2.1.

Listing 2.1: Log output from data flow example.

```
>> log stream | grep bluetooth | grep openConnection
2019-12-21 11:44:35.812766+0100 0x1368e     Default      0
    x277cb               1538    0     System Preferences: (
    IOBluetooth) [com.apple.bluetooth:IOBluetoothDevice] [
    openConnection] self=0x60000177bd80 target=0x600003793a50
     pageTimeoutValue=0x0000 authenticationRequired=0
    allowRoleSwitch=0
```

IOBLUETOOTH Finally, `openConnection` calls a private function of `IOBluetooth`, called `-[IOBluetoothHostController BluetoothHCIReadPageTimeout:]` that calls `_BluetoothHCIDispatchUserClientRoutine` to communicate with `IOKit`.

IOKIT In the last step, `IOKit` receives the command in `_IOConnectCallStructMethod` and sends it to the `IOBluetoothFamily` driver via Mach messages.

2.2 KERNEL EXTENSIONS AND FRAMEWORKS IN MACOS

BUNDLES In macOS, there is the concept of *Bundles*. They are specific directories with a well-defined structure and a file extension [1]. Bundles stem from the macOS predecessor NeXTSTEP and are also used in OPENSTEP, GNUSTEP and iOS. The location of the binary of a Bundle varies depending on its type and using the file extension, the operating system determines what kind of bundle it is dealing with.

For example, a macOS Application is a bundle with the *.app* extension, so when double-clicking it in the Graphical User Interface (GUI), the Operating System (OS) knows that it has to traverse a few layers into the folder structure, find the binary and execute it. Similar to Application bundles, there are also Framework and Kext bundles. Frameworks are included and used by another piece of software, while Kexts extend the kernel.

Frameworks have the executable in `Framework.framework/Versions/A/Framework`, Kernel Extensions have it in `Kext.kext/Contents/MacOS/Kext` and Applications have their executable in `Application.app/Contents/MacOS/Application`.

Even on Windows, Kernel Extensions do exist under the name "kernel-mode driver".

KERNEL EXTENSIONS Kexts exist on most operating systems and there are lots of guides for their development, like *The Linux Kernel Module Programming Guide* [12].

In macOS, every Kernel Extension has to be signed with a *"Developer ID Certificate for signing kexts"* [2], otherwise, it will not be loaded by the operating system. Nowadays, the only method to run them unsigned is to disable macOS' System Integrity Protection (SIP) [2], which is generally very unrecommended due to security reasons like accidentally running a malicious, unsigned Kext. While SIP was introduced in macOS 10.11 El Capitan, even in prior versions of macOS (then called OS X), Kernel extensions had to be signed and the boot flag `kext-dev-mode=1` had to be set [2] to run unsigned ones, which helped developers create their Kexts.

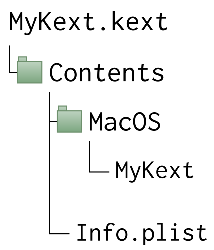

Figure 2.2: Basic structure of a macOS Kext Bundle.

```
IOBluetoothFamily.kext
└ Contents
  ├ _CodeSignature
  │ └ CodeResources
  ├ Info.plist
  ├ MacOS
  │ └ IOBluetoothFamily
  ├ PlugIns
  │ ├ BroadcomBluetooth20703USBTransport.kext
  │ ├ BroadcomBluetoothHostControllerUSBTransport.kext
  │ ├ CSRBluetoothHostControllerUSBTransport.kext
  │ ├ CSRHIDTransitionDriver.kext
  │ ├ IOBluetoothHostControllerPCIeTransport.kext
  │ ├ IOBluetoothHostControllerTransport.kext
  │ ├ IOBluetoothHostControllerUARTTransport.kext
  │ ├ IOBluetoothHostControllerUSBTransport.kext
  │ ├ IOBluetoothSerialManager.kext
  │ └ IOBluetoothUSBDFU.kext
  ├ Resources
  │ └ ...
  └ version.plist
```

Figure 2.3: Structure of the `IOBluetoothFamily.kext`.

Kernel extensions are sometimes referred to as Loadable Kernel Modules (LKM). In many cases, these are device drivers, for example for Bluetooth and Wi-Fi chips. Moreover, they can add support for new filesystems and system calls [16]. Generally Kernel extensions can be loaded and unloaded while the system is running. In macOS however there are a few specific locations in the file system that lead to automatic loading of the extension at boot time. Since they work at a very low level inside of the OS, they only have access to certain headers that are available in the kernel, so Kexts have to be compiled statically and without linking to the C standard library. [5]

Kexts shipped with macOS reside in /System/Library/Extensions, while applications should store their Kexts in /Library/Extensions.

Listing 2.2 contains macOS utilities for working with Kernel Extensions.

Listing 2.2: BSD utilities for Kernel Extensions available in macOS

```
kext_logging, kextcache, kextd, kextfind, kextlibs, kextload
    , kextstat, kextunload, kextutil.
```

In the following section, we will analyse which Kernel extensions are the main Bluetooth device drivers. We will see that macOS contains lots of Kexts, while only a few are loaded depending on hardware and macOS version. For example, there are different Kexts to deal with Bluetooth chips that are attached via USB, UART and PCIe.

To give programmers (also their own ones) a universal programming interface, Apple provides the `IOBluetooth` framework with public and private methods and the `IOKit` framework which also deals with some Bluetooth functionality.

2.2.1 *IOBluetooth.framework and IOKit.framework*

By using `kextstat` we can find the bundle identifiers of loaded Kernel extensions that have Bluetooth in their name. We can see that on lower levels of macOS, Bluetooth is categorised as part of `IOKit`, while those Kexts are not specifically part of the `IOKit` framework. `IOKit` is just part of the bundle identifier.

Listing 2.3: Loaded Bluetooth Kexts (MacBook Pro (MBP) 15,3 - macOS 10.15.1)

```
>> kextstat | grep -i bluetooth | awk '{print $6}'
com.apple.iokit.IOBluetoothPacketLogger
com.apple.iokit.IOBluetoothFamily
com.apple.iokit.IOBluetoothHostControllerTransport
com.apple.iokit.IOBluetoothHostControllerUARTTransport
com.apple.driver.IOBluetoothHIDDriver
com.apple.driver.AppleHSBluetoothDriver
com.apple.iokit.IOBluetoothSerialManager
```

Listing 2.4 combines `kextstat` and `kextfind` to display the locations of the previously found Kernel extensions and apparently many of them are part of the `IOBluetoothFamily.kext`. As we will see later, this is the main Bluetooth driver in macOS and itself contains Kexts as plugins.

Listing 2.4: Loaded Kext locations (MBP 15,3 - macOS 10.15.1)

```
>> for i in $(kextstat | grep -i bluetooth | awk '{print $6
    }'); do kextfind -b $i; done
/System/Library/Extensions/IOBluetoothHIDDriver.kext
/System/Library/Extensions/AppleTopCase.kext/Contents/
    PlugIns/AppleHSBluetoothDriver.kext
/System/Library/Extensions/IOBluetoothFamily.kext
/System/Library/Extensions/IOBluetoothFamily.kext/Contents/
    PlugIns/IOBluetoothPacketLogger.kext
/System/Library/Extensions/IOBluetoothFamily.kext/Contents/
    PlugIns/IOBluetoothHostControllerTransport.kext
```

```
/System/Library/Extensions/IOBluetoothFamily.kext/Contents/
    PlugIns/IOBluetoothHostControllerUARTTransport.kext
/System/Library/Extensions/IOBluetoothFamily.kext/Contents/
    PlugIns/IOBluetoothSerialManager.kext
```

While Listing 2.3 shows all of the currently loaded Bluetooth related Kernel extensions, we can see that most of them reside in the *PlugIns* subfolder of the `IOBluetoothFamily.kext` bundle (Listing 2.5), so looking in there reveals even more Kernel extensions that can be loaded on demand.

Listing 2.5: Kexts inside `IOBluetoothFamily` (MBP 15,3 - macOS 10.15.1)

```
>> ls /System/Library/Extensions/IOBluetoothFamily.kext/
    Contents/PlugIns
BroadcomBluetooth20703USBTransport.kext
BroadcomBluetoothHostControllerUSBTransport.kext
CSRBluetoothHostControllerUSBTransport.kext
CSRHIDTransitionDriver.kext
IOBluetoothHostControllerPCIeTransport.kext
IOBluetoothHostControllerTransport.kext
IOBluetoothHostControllerUARTTransport.kext
IOBluetoothHostControllerUSBTransport.kext
IOBluetoothPacketLogger.kext
IOBluetoothSerialManager.kext
IOBluetoothUSBDFU.kext
```

In this thesis, we discovered that the 2019 MacBook Pro 15,4 and iPhone 11 line are the first Apple products to use PCIe communication with the Bluetooth chip, whereas before UART or USB were used. USB can be either internal or external, in case Bluetooth is provided by a USB dongle. The main advantage of PCIe is speed and is most noticeable when having multiple Bluetooth devices connected at the same time.

The entry-level 2019 MacBook Pro (model 15,4) uses PCIe, whereas the more expensive "Four Thunderbolt 3 Ports" (15,2) and 15-inch (15,3) models still use UART.

By examining Table 2.1 and Table 2.2 we can figure out which Kernel extensions are loaded on every Mac.

	2019			2017	2014 & 2013	
Kext Name	**MBP 16,1** 10.15.1	**MBP 15,4** 10.15.1	**MBP 15,3** 10.15.1	**MBP 14,2** 10.15.1	**MBP 11,1** 10.15.1	**MBP 11,1** 10.14.6
BroadcomBluetooth20703USBTransport.kext	✓	✓	✓	✓	✓	✓
BroadcomBluetoothHostControllerUSBTransport.kext	✓	✓	✓	✓	✓	✓
CSRBluetoothHostControllerUSBTransport.kext	✓	✓	✓	✓	✓	✓
CSRHIDTransitionDriver.kext	✓	✓	✓	✓	✓	✓
IOBluetoothHostControllerPCIeTransport.kext	✓	✓	✓	✓	✓	✓
IOBluetoothHostControllerTransport.kext	✓	✓	✓	✓	✓	✓
IOBluetoothHostControllerUARTTransport.kext	✓	✓	✓	✓	✓	✓
IOBluetoothHostControllerUSBTransport.kext	✓	✓	✓	✓	✓	✓
IOBluetoothPacketLogger.kext	✓	✓	✓	✓	✓	
IOBluetoothSerialManager.kext	✓	✓	✓	✓	✓	✓
IOBluetoothUSBDFU.kext	✓	✓	✓	✓	✓	✓
IOBluetoothHIDDriver.kext	✓	✓	✓	✓	✓	✓
AppleHSBluetoothDriver.kext	✓	✓	✓	✓	✓	✓
IOBluetoothFamily.kext	✓	✓	✓	✓	✓	✓

Table 2.1: List of Bluetooth Kexts that are available in different MBP models.

IOBluetoothFamily.kext is the Bluetooth driver on macOS. The other two Kexts loaded on every system and OS are part of IOBluetoothFamily as well since they reside in its PlugIns directory.

Table 2.1 shows all directly Bluetooth-related Kernel extensions. Independently from the Mac model and macOS version, this list is almost the same on every computer that was tested. As mentioned before, only a subset of these extensions are actually loaded and active—that's where things start to differ a lot depending on computer and system version.

	2019			2017	2014 & 2013	
Kext Name	**MBP 16,1** 10.15.1	**MBP 15,4** 10.15.1	**MBP 15,3** 10.15.1	**MBP 14,2** 10.15.1	**MBP 11,1** 10.15.1	**MBP 11,1** 10.14.6
BroadcomBluetooth20703USBTransport.kext						
BroadcomBluetoothHostControllerUSBTransport.kext					✓	✓
CSRBluetoothHostControllerUSBTransport.kext						
CSRHIDTransitionDriver.kext						
IOBluetoothHostControllerPCIeTransport.kext		✓				
IOBluetoothHostControllerTransport.kext	✓	✓	✓	✓	✓	✓
IOBluetoothHostControllerUARTTransport.kext	✓		✓	✓		
IOBluetoothHostControllerUSBTransport.kext					✓	✓
IOBluetoothPacketLogger.kext	✓	✓	✓	✓	✓	
IOBluetoothSerialManager.kext	✓	✓	✓	✓	✓	✓
IOBluetoothUSBDFU.kext						✓
IOBluetoothHIDDriver.kext	✓	✓	✓	✓		
AppleHSBluetoothDriver.kext	✓	✓	✓	✓		
IOBluetoothFamily.kext	✓	✓	✓	✓	✓	✓

Table 2.2: List of Bluetooth Kexts that are loaded in different MBP models.

From Table 2.2 we can deduce that the following Kexts are loaded on every machine:

- `IOBluetoothHostControllerTransport.kext`
- `IOBluetoothSerialManager.kext`
- `IOBluetoothFamily.kext`

The first two Kernel Extensions are actually part of `IOBluetoothFamily` and can be found in its `Contents/PlugIns/` subdirectory. All of these three Kexts are loaded on every Mac directly at startup and provide the core Bluetooth driver functionality as we found during disassembly. Only the data transport is delegated to various Bluetooth transport Kexts depending on the technology that Apple chose for each Mac.

Our study shows that Macs released since 2017 use UART to communicate with their internal Bluetooth chips, so they load `IOBluetoothHostControllerUARTTransport.kext` while the older Macs use USB, thus loading `IOBluetoothHostControllerUSBTransport.kext` and additionally `BroadcomBluetoothHostControllerUSBTransport.kext`.

The MacBook Pro 15,4 uses PCIe, even though it is the base model 13" version and more expensive models still use UART (including the latest 16" MacBook Pro 16,1).

`IOBluetoothHIDDriver.kext` and `AppleHSBluetoothDriver.kext` are loaded on all models using UART or PCIe communication.

It appears that `IOBluetoothPacketLogger.kext` is only present and loaded on systems running macOS Catalina 10.15.x, independently from the computer's launch date and connection with the Bluetooth chip.

2.3 GRAND CENTRAL DISPATCH AND DISPATCHQUEUES

Since the HCI controller delivers responses asynchronously via events, we have to take a look at multithreading. Especially the performance of *InternalBlue* commands like `dumpmem`, where HCI commands are sent repeatedly in quick succession, would otherwise suffer greatly as the sending command blocks the thread until its result has arrived.

Since OS X Snow Leopard, which was released in 2009, Apple offers a rich high-level cross-platform API for threading. Grand Central Dispatch (GCD) is now available for macOS, iOS, tvOS and even on watchOS and makes it easy to write multithreaded code, regardless of device or platform. It acts as an abstraction layer from the actual threads in the CPU. So another point to consider is that regular Mac and iOS applications, running in a sandbox, do not really have any information about other running processes and available resources. When creating and managing threads manually, one might create too many or too few threads leading to suboptimal performance due to the assumptions that have to be made. Instead, GCD operates at the system level so it can perfectly estimate how many threads to create and how to manage them efficiently.

A regular Mac application runs on the main thread, so processing-heavy operations will freeze the application while they are being performed and make it feel unresponsive. This is the typical use case for so-called `DispatchQueues`. From a programmer's standpoint, CPU-heavy or asynchronous work like network calls or in our case requests to an external chip are scheduled onto such a `DispatchQueue` and GCD then decides when and how to execute that work using its pool of threads. By default, there are multiple background queues, which is also what we used in the macOS port that is part of this thesis, but custom queues can be created through the GCD API. All tasks in a queue are started in First-In-First-Out (FIFO) order, even though of course they might end out-of-order due to longer processing times.

User Interface must be refreshed on the main thread on all Apple devices, so asynchronously finishing tasks have to dispatch any User Interface (UI) changes to the main thread. Long-running tasks should be dispatched to a background thread to prevent UI lockup.

GCD offers semaphores in case that a specific task has to be completed before another one is started and `DispatchGroups` notify an application when all tasks in that group have been completed.

One last thing that is especially important on mobile devices is power management. As mentioned before, sandboxed applications cannot really know how much power the CPU currently draws, how much it is loaded and depending on API availability might not even know the current battery status. GCD instead has the full overview and, e.g., when an iPhone is in power saving mode it can use different scheduling strategies to make sure it does not draw too much power.

Listing 2.6 shows how it looks in practice. First of all, a queue is selected. In this case, the preconfigured `.background` queue that runs on the background thread is chosen. Then, in the *closure* that is delimited by curly brackets, follows the work to be performed asynchronously.

Listing 2.6: Dispatch work onto background thread

```
DispatchQueue.global(qos: .background).async {
  // long running task
}
```

3 RELATED WORK

The following chapter presents other work in the field that is related to my thesis.

Section 3.1 introduces the *InternalBlue* project that was ported to macOS as an integral part of this thesis, while Section 3.2 helped a lot—especially in the beginning—to start getting an understanding of the Bluetooth architecture in macOS. This makes these two projects mostly related to the codebase of *InternalBlue*'s Mac port.

Section 3.3 presents two projects that delve into other areas of Bluetooth-related reversing in the Apple ecosystem. In the first one, Apple Wireless Direct Link (AWDL) was reversed and implemented as open-source software as well as *AirDrop* and more. The second project is about reverse-engineering Apple's Handoff and Universal Clipboard features.

3.1 INTERNALBLUE

As previously mentioned, the core contribution of this work is a macOS port of the *InternalBlue* project [7, 8].

InternalBlue is a Bluetooth experimentation framework that allows to research Bluetooth firmware on the Broadcom chips that are built into most off-the-shelf consumer electronics. Since trying to intercept wireless Bluetooth signals with antennas and decoding that data is unviable e.g. due to frequency hopping in a wide frequency range, *InternalBlue* works at the Host-Controller Interface (HCI) layer. As HCI is the lowest level of the Bluetooth chip accessible by the host Operating System (OS), arbitrary commands and events can be exchanged with the chip through *InternalBlue*.

There is an interactive console with a variety of commands that then in turn call HCI commands on the Bluetooth controller. For example, a single HCI `readMem` command can only read 251 bytes of memory at a time, but *InternalBlue* offers a `dumpmem` command which runs `readMem` in a loop and stitches the resulting event data together into a binary file. Similarly to `dumpmem`, there are other very helpful features like writing to memory, disassembling program code in Read-Only Memory (ROM) and Random Access Memory (RAM) and executing parts of the memory on the fly. *InternalBlue* also offers an interface for *Wireshark* and monitors connections that are made to and from the attached Bluetooth chip.

On Broadcom Bluetooth chips, RAM is executable, so custom assembly snippets can be patched in and executed using InternalBlue.

InternalBlue runs on Linux devices and with this work on macOS. Through Android Debug Bridge (ADB), it is possible to attach to an

Android devices' Bluetooth chip and in a similar fashion, a jailbroken iOS devices' chip can be accessed through a wired connection.

3.2 SENDING ARBITRARY HCI COMMANDS

InternalBlue needs a way to send HCI commands (for example in the aforementioned `dumpmem` loop) as byte arrays and expects to get byte arrays back via socket communication. There are projects that already use the private Application Programming Interface (API) [9–11]. Their capabilities were too limited for our purposes but served as a starting point (Chapter 4).

3.3 OTHER BLUETOOTH-RELATED APPLE REVERSING

The Secure Mobile Networking Lab at TU Darmstadt is reverse-engineering Apple's Bluetooth stacks, protocols and features for security research purposes.

So far, Apple's proprietary wireless protocol AWDL [15], which is used for *AirDrop*, *AirPlay* and Apple Watch Auto Unlock, was reverse-engineered. An open-source implementation called Open Wireless Link (OWL) [14] was created that runs on macOS and Linux. Based on OWL, *OpenDrop* [13] was created: an open-source *AirDrop* implementation.

The magic behind Apple's Handoff and Continuity features was also unveiled and presented at Macoun 2019 [6], the largest European macOS and iOS developer conference.

3.3.1 *Apple Wireless Direct Link (AirDrop, AirPlay)*

AWDL is an ad hoc protocol that uses Wi-Fi technology to directly communicate between end-user devices, also in absence of any Wi-Fi access points nearby. It is used in most of Apple's proprietary wireless features like *AirDrop*, which allows to directly send images and other files from one Apple device to another as well as *AirPlay*, that enables music and video streaming from an iOS or macOS device to the Apple TV. Since AWDL is proprietary and undocumented, security and other issues with the protocol had not been studied yet, even though it is deployed on over a billion devices. The researchers uncovered the macOS Wi-Fi driver architecture, reverse-engineered AWDL and performed various tests and security analyses. They finally reimplemented AWDL as well as *AirDrop* as open-source software.

3.3.2 *Continuity: Handoff and Universal Clipboard*

Location and user behaviour can be tracked by listening for BLE announcements and following the unencrypted IV counter.

Handoff and Universal Clipboard are two Continuity features that send Bluetooth Low Energy (BLE) advertisements with each user interaction such as copying text or using an App that supports Handoff. Apple devices are typically very verbose over BLE, but the content of these advertisements is partially hashed and they are all encrypted using AES-GCM, so no information about the actions of users can be retrieved by an attacker. More details about privacy and security have been presented at Macoun 2019 [6].

4 IMPLEMENTATION

This chapter deals with the implementation of full Host-Controller Interface (HCI) access into a macOS port for *InternalBlue*. Section 4.1 explains how we were able to circumvent Apple's high-level Application Programming Interface (API) and work directly with the lowest interface to the Bluetooth chip that Apple themselves use. In Section 4.2, we show how to package the code providing this low-level access in a way that makes it accessible from our Python project *InternalBlue* and Section 4.3 includes the changes that had to be made in the Python codebase to make it work on macOS. Finally, Section 4.4 covers the integration of our custom framework into *InternalBlue*.

4.1 ACCESSING AN UNDOCUMENTED BLUETOOTH INTERFACE

By decompiling `/usr/bin/bluetoothd` using *Hopper v4*, we quickly discovered that Apple implemented all methods of the official Bluetooth specification as private methods of `IOBluetoothHostController`. To access them from an Objective-C project it suffices to import a header file declaring all of the undocumented methods. In this way, one can call directly all methods of the Bluetooth specification.

Only the driver IOBluetoothFamily needs elevated privileges. It is loaded when the OS boots and is accessed by IOKit over Mach messaging.

Apple's private implementation of `BluetoothHCISendRawCommand` allows to handle HCI commands as byte arrays, which perfectly matches the approach in *InternalBlue*.

Root access is not needed, so (unlike on Linux) an unprivileged user account is sufficient to access the Bluetooth chip via HCI.

To retrieve the events with the results of the sent messages, there is an interface called `IOHostControllerDelegate`. Upon creation, an object conforming to this interface automatically receives all of the `IOKit` events regarding Bluetooth.

4.2 CUSTOM IOBLUETOOTHEXTENDED FRAMEWORK

All of the steps mentioned—from the reversed header to the `IOHostControllerDelegate` implementation—are done in Objective-C. *InternalBlue*, however, is written in Python and while using *pyobjc* could allow writing Objective-C code in Python, a completely *pyobjc* based implementation would be difficult to maintain due to lack of documentation and infrequent use by the developer community. Additionally, data type incompatibilities and memory access differences between the two programming languages increase the difficulty of creating a stable application.

We therefore decided to minimise the amount of *pyobjc* usage and reduced it to the import of our IOBluetoothExtended (IOBE) framework and the instantiation of an `IOBE` object.

IOBluetoothExtended originally only extended IOBluetooth by exposing its private methods and later evolved to a middle layer between InternalBlue and IOBluetooth.

InternalBlue is based on sockets, which simplifies porting to new operating systems. There is a `core` in *InternalBlue* that is subclassed for each system. In case of the `macoscore`, an instance of the `IOBE` object, which is declared in the framework, is created. In the initialiser, the port numbers for the input (`s_inject`) and output (`s_snoop`) sockets are passed and no more Objective-C functions or objects have to be accessed from Python code after that since all communication happens through the aforementioned sockets.

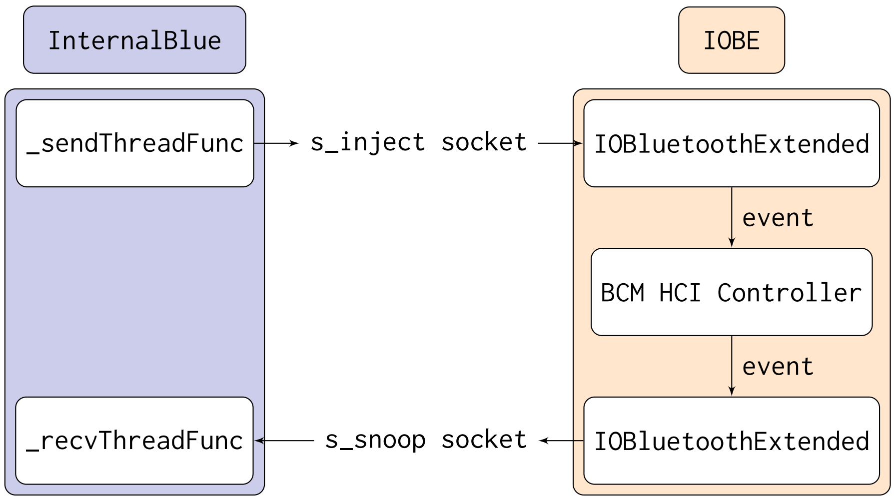

Figure 4.1: Interaction between *InternalBlue* (Python) and *IOBE* (Objective-C).

4.2.1 *HCI communication*

According to Apple's `IOBluetooth` specification, the `IOBluetoothHostController` object is *"[...] a representation of a Bluetooth Host Controller Interface that is present on the local computer (either plugged in externally or available internally)."* [3]

Only private methods of the `IOBluetooth` API allow to send HCI and Asynchronous Connection-Less (ACL) commands as simple byte arrays filled with values from the Bluetooth specification.

Listing 4.1 shows the crucial private methods that are used in our implementation.

Listing 4.1: Crucial private methods

```
int BluetoothHCIRequestCreate(uint32_t *request,
    int timeout, void* arg3, size_t arg4);

int BluetoothHCIRequestDelete(uint32_t request);

int BluetoothHCISendRawCommand(uint32_t request,
    void *commandData, size_t commmandSize);

int BluetoothHCISendRawACLData(void *commandData,
    size_t commandSize, uint32_t handle, uint32_t request);

int BluetoothHCIDispatchUserClientRoutine(
    struct IOBluetoothHCIDispatchParams *arguments,
    unsigned char *returnValue, size_t *returnValueSize);
```

We found these functions in the `IOBluetooth` binary from the `IOBluetooth.framework` using *Hopper v4* since the function names are not obfuscated.

The code in Listing 4.2 is used as part of the custom-made `IOBluetoothExtended` framework to be able to interact with the HCI Controller via macOS.

Listing 4.2: sendArbitraryCommand function

```
+ (void) sendArbitraryCommand:
  (uint8_t [])arg1 len:(uint8_t)arg2 {

  // First of all, the HCI command bytes are
  // copied into a local variable on the heap
  NSData *data = [NSData dataWithBytes:arg1 length:arg2];
  uint8_t *command = calloc(arg2, sizeof(uint8_t));
  memcpy(command, [data bytes], arg2);

  // The request ID is initialised
  // with an arbitrary integer
  BluetoothHCIRequestID request = 0;

  // Get a request ID from bluetoothd
  // We also tell it the timeout in ms (1000)
  int error = BluetoothHCIRequestCreate(
    &request, 1000, nil, 0);
  if (error) {
    BluetoothHCIRequestDelete(request);
    printf("Couldn't create error: %08x\n", error);
  }

  // The length parameter that the HCI Controller
  // needs to know, is embedded in the command
  // itself in all longer HCI commands
  size_t commandSize = 3;
  if (arg2 > 2) {
    commandSize += command[2];
  }
```

```
  // Finally, all parameters are set and the request
  // can be sent to the chip via this third important
  // private method
  error = BluetoothHCISendRawCommand(
    request, command, commandSize);

  if (error) {
    BluetoothHCIRequestDelete(request);
    printf("Send HCI command Error: %08x\n", error);
  }

  sleep(0x1);
  // Delete the request, just some cleanup
  BluetoothHCIRequestDelete(request);
}
```

4.2.2 *BSD UDP server*

Since `IOBluetoothExtended` acts as a middle-layer between *InternalBlue* and the macOS Bluetooth stack and *InternalBlue* tries to send its command to a socket, the framework also includes a User Datagram Protocol (UDP) client and server mechanism. It accepts commands and sends back all asynchronous responses from the chip. Another benefit of this approach is that applications like *Wireshark* can be used to view the live traffic and decode or save it to disk.

InternalBlue shows the ports to put in Wireshark's filter settings on startup when executed on macOS.

While many socket frameworks exist, like Apple's `Network` framework, we decided to use standard Berkeley Software Distribution (BSD) socket system calls like `bind`, `recvfrom` and `sendto`. This approach uses only a few more lines of code, but it is more maintainable and easier to understand for anyone who has socket knowledge, without learning the framework or weird Objective-C concepts.

In Listing A.1, we can see that the `startupServer()` function begins with the creation of a UDP server that listens on the specified `s_inject` socket.

After successfully binding to the desired port, in Listing A.2 an infinite loop is started on the background thread and the framework starts waiting for data with the `recvFrom` system call.

Finally, Listing A.3 contains the transmission of HCI commands to the chip. When a command arrives from *InternalBlue*, `HCIDelegate` decodes the message and—again on the background queue—passes it as a byte array to the static function `sendArbitraryCommand()` of the `HCICommunicator` which has been examined in Listing 4.2. This class imports the aforementioned reverse engineered header file that declares all of the private methods that Apple implemented into `IOBluetooth`. It can then call `BluetoothHCISendRawCommand(uint32_t request, void *commandData, size_t commmandSize)` and send arbitrary commands to the HCI controller using a pointer to a byte array as the second argument.

4.3 INTERNALBLUE MACOS CORE

InternalBlue has a modular, expansible structure which enables the integration of more platforms. A lot of *InternalBlue*'s functionality lies in `core.py`, which is subclassed to override device- or OS-dependent functions. The new core is then added as a launch option in `cli.py`.

For the macOS port, `macoscore.py` was created—detection of the platform is done automatically by `cli.py`, so no further parameters are necessary and `macoscore.py` is chosen by default.

Listing 4.3 shows how our custom framework is imported into the `macoscore`.

Listing 4.3: macOS Core Python Imports

```
import objc
import os
filepath = os.path.dirname(os.path.abspath(__file__))
objc.initFrameworkWrapper("IOBluetoothExtended",
  frameworkIdentifier=
    "de.tu_darmstadt.seemoo.IOBluetoothExtended",
  frameworkPath=objc.pathForFramework(filepath+"/../macos-
      framework/IOBluetoothExtended.framework"),
  globals=globals())
```

The most important override of `macoscore.py` is the function (Listing 4.4) `def _setupSockets(self)`. First of all, random socket numbers are determined to avoid collisions that would otherwise happen especially often when shutting *InternalBlue* down and opening it up again repeatedly, since it usually takes a bit of time for them to be released by the OS.

Then, one of the two local UDP servers is started, which is later used to receive events from the IOBE framework. Afterwards, the socket to send commands to IOBE and thus to the chip is created.

Lastly, an instance of IOBE is first allocated and then initialised with the custom initialiser where input and output port numbers are passed. They are passed as strings since the translation from a Python string to an Objective-C string is less problematic than from a Python *number* to a strongly typed Objective-C numerical type.

In our experiments, we observed that the initialisation of `IOBE` could take up to 100ms. In order to ensure that the UDP server was up when the first commands are transmitted, we introduced a delay of 500ms before declaring the socket setup as completed.

Listing 4.4: macOS Core _setupSockets function

```
def _setupSockets(self):
  self.hciport = random.randint(60000, 65535-1)
```

```
    log.debug("_setupSockets: Selected random ports snoop=%d
        and inject=%d" % (self.hciport, self.hciport + 1))
    log.info("Wireshark configuration (on Loopback interface):
         udp.port == %d || udp.port == %d" % (self.hciport,
        self.hciport + 1))

    # Create s_snoop socket
    self.s_snoop = socket.socket(socket.AF_INET, socket.
        SOCK_DGRAM)
    self.s_snoop.setsockopt(socket.SOL_SOCKET, socket.
        SO_REUSEADDR, 1)
    self.s_snoop.bind(('127.0.0.1', self.hciport))
    self.s_snoop.settimeout(0.5)
    self.s_snoop.setblocking(True)

    # Create s_inject
    self.s_inject = socket.socket(socket.AF_INET, socket.
        SOCK_DGRAM)
    self.s_inject.settimeout(0.5)
    self.s_inject.setblocking(True)

    # Create IOBluetoothExtended Object that listens for
    # commands, sends them to the Bluetooth chip and replies
    # via UDP socket.
    self.iobe = IOBE.alloc().initWith_and_(str(self.hciport+1)
        , str(self.hciport))
    time.sleep(0.5)

    return True
```

The rest of the initialisation takes place in the `IOBE` initialiser, as seen in Listing 4.5.

4.4 FRAMEWORK AND INTERNALBLUE INTEGRATION

Even bluetoothd registers as an IOBluetoothHost-Controller delegate to receive specific events.

Listing 4.5 shows the `IOBE` object, which upon creation from the Python code with input and output ports, creates an `HCIDelegate` object and allocates it to a saved pointer to the `IOBluetoothHostController` (HCI Controller). As we will see in Listing 4.6, the initialisation of the `HCIDelegate` launches the UDP server and hooks into the `UIKit` events for `IOBluetooth`. It receives all Bluetooth HCI events in the implementation of the `@objc(BluetoothHCIEventNotificationMessage:inNotificationMessage:)` function. To make its handling a bit easier, `HCIDelegate` was implemented in the latest version of Swift 5, so the code is more readable and maintainable for future contributors to this project. The `NSRunLoop` call ensures that the function does not exit immediately, which would also kill the loop waiting for commands and the UDP server.

Listing 4.5: `IOBE` initialisation

```
#import "IOBE.h"
#import "HCIDelegate.h"
```

```
@implementation IOBE

- (id) initWith:(NSString *)inject and:(NSString*)snoop {
  if (self = [super init]) {
    dispatch_async(dispatch_get_global_queue(
      DISPATCH_QUEUE_PRIORITY_BACKGROUND, 0), ^{
        self->controller =
          IOBluetoothHostController.defaultController;
        self->delegate =
          [[HCIDelegate alloc] initWith:inject and:snoop];
        self->controller.delegate = self->delegate;

        [[NSRunLoop currentRunLoop] run];
    });
  }
  return self;
}
```

The HCIDelegate's initialiser (Listing 4.6) mainly saves the ports and starts one of its Swift functions startupServer(), which we have examined in Section 4.2.2. The second main function of this object is to receive notifications about all Bluetooth HCI Events and report them back to *InternalBlue* via the s_snoop socket.

Listing 4.6: HCIDelegate initialisation

```
#import "HCIDelegate.h"
#import "IOBluetoothExtended/IOBluetoothExtended-Swift.h"

@implementation HCIDelegate

Boolean exit_requested = false;

- (id) initWith:(NSString *)inject and:(NSString*)snoop {
  if (self = [super init]) {
    self.inject = inject;
    self.snoop = snoop;
    self.hostname = @"127.0.0.1";
    [self initServer];
  }
  return self;
}

...

@objc public func initServer() {
  self.startupServer()
}
```

Listing 4.7 shows the processing of HCI events performed by the HCIDelegate before transporting them to *InternalBlue*. For example, empty data packets have to be discarded.

Another detail that complicates implementation is that HCI event payloads are restructured by `IOBluetoothHostController::ProcessEventDataWL` in the Bluetooth driver, which results in abstracted HCI events that are returned as objects of type `IOBluetoothHCIEventNotificationMessage`. To regain a byte array, the data has to be stitched together.

Listing 4.7: Receiving Bluetooth HCI events (part 1/2)

```
@objc(BluetoothHCIEventNotificationMessage:
    inNotificationMessage:)
public func bluetoothHCIEventNotificationMessage(_
    controller: IOBluetoothHostController,
  in message: UnsafeMutablePointer<
      IOBluetoothHCIEventNotificationMessage>) {

  let opcode = message.pointee.dataInfo.opcode
  let data = IOBluetoothHCIEventParameterData(message)
  if opcode == 0 { return }

  let dataInfo = message.pointee.dataInfo
  let opcod1 = String(format:"%02X", dataInfo.opcode)
  let opcod2 = Array(repeating: "0", count: 4-opcod1.count)
      + Array(opcod1)
  if opcod2.count < 4 { return }
  let opcod3 = "\(opcod2[2])\(opcod2[3])\(opcod2[0])\(opcod2
      [1])"

  var result = "04"
  result.append(String(format:"%02X", dataInfo._field7))
  result.append("\(String(format:"%02X", dataInfo.
      parameterSize+3))")
  result.append("01\(opcod3)")
  result.append(data.hexEncodedString())

  if result.count < 8 { return }

  let h = NWEndpoint.Host(self.hostname as String)
  let s = NWEndpoint.Port(self.snoop as String)
```

Apple's `IOBluetoothHCIEventNotificationMessage` object's byte order does not conform entirely to the Bluetooth standard for specific commands. For example, *Read Local Version Information* (`0x1001`), *Connection Complete* (`0x0405 / 0x0409`) and *Disconnection complete* (`0x0406`) deliver the wrong byte orders. For this reason we had to manually remap them (Listing 4.8).

Here's where Swift, with easier and shorter String and Array operations and type conversions, comes into play. After remapping the events into default, spec-conforming format, they can be sent to *InternalBlue* via the `sendOverUDP()` function so that they are interpreted correctly.

Listing 4.8: Receiving Bluetooth HCI events (part 2/2)

```
    // HCI_Read_Local_Version_Information
    if opcode == 0x1001 {
      var temp = ""
      for i in [0,1,2,3,4,5,9,8,14,15,12,6,7,10,11] {
        temp.append(result[i*2])
        temp.append(result[i*2+1])
      }
      self.sendOverUDP(data: temp.hexadecimal!, h, s!)
    }
    // HCI_Connection_Complete
    else if opcode == 0x0405 || opcode == 0x0409 {
      let orig = data.hexEncodedString()
      var temp = "0403"
      for i in [8,9,0,1,7,6,5,4,3,2] {
        temp.append(orig[i*2])
        temp.append(orig[i*2+1])
      }
      if temp.count != 24 { return }
      self.sendOverUDP(data: temp.hexadecimal!, h, s!)
    }
    // HCI_Disconnection_Complete
    else if opcode == 0x0406 {
      let orig = data.hexEncodedString()
      if orig.count == 0 { return }
      var temp = "040504"
      for i in [2,1,0] {
        temp.append(orig[i*2])
        temp.append(orig[i*2+1])
      }
      self.sendOverUDP(data: temp.hexadecimal!, h, s!)
    }
    else {
      let temp = result.hexadecimal!
      if temp.count >= 8 {
        self.sendOverUDP(data: temp, h, s!)
      }
    }
  }
```

In Table 4.1 we summarise the difference between the data from the Bluetooth driver and the requirements from *InternalBlue*. We were able to determine the bytes that have to be swapped and retrieved their respective indexes. Sometimes, some bytes were also missing completely, but were static, so `HCIDelegate` just adds them.

Index					0	1	2	3	4	5	6	7	8	9	10	11
HCI_Read_Local_Version_Information																
IOBluetooth					01	10	0F	00	06	00	96	41	06	00	67	24
					Opcode		Manufacturer_Name		HCI_Version		LMP_Subversion		LMP_Version		HCI_Revision	
Expected	04	0E	0C	01	01	10	00	06	67	24	06	0F	00	96	41	
					Opcode		Status	HCI_Version	HCI_Revision		LMP_Version	Manufacturer_Name		LMP_Subversion		
HCI_Connection_Complete																
IOBluetooth					0d	00	90	E1	7B	63	B5	8D	01	00		
					Connection_Handle		BD_ADDR (left to right)									
Expected			04	03	01	00	0d	00	8D	B5	63	7B	E1	90		
							Connection_Handle		BD_ADDR (right to left)							
HCI_Disconnection_Complete																
IOBluetooth					0d	00	13	00								
					Connection_Handle		Reason	Status								
Expected		04	05	04	00	0d	00	13								
					Status	Connection_Handle		Reason								

Table 4.1: Mappings between the Kext output and *InternalBlue* as determined manually for some HCI commands from the Bluetooth specification.

4.5 ACL COMMUNICATION IN MACOS

Most data, especially high quality audio, is transmitted to connected Bluetooth devices using the ACL protocol. Also tethering via Bluetooth works through ACL, for example to use a phone's mobile data plan to get internet on a laptop. ACL packets are in turn transmitted through Logical Link Control and Adaption Protocol (L2CAP).

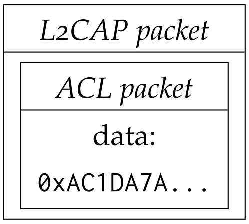

Figure 4.2: Basic structure of an ACL packet.

4.5.1 *Reversing IOBluetooth*

The disassemblers *Hopper v4* and *Ghidra* were used to reverse engineer how ACL is done via `IOBluetooth`. Targeted binaries were `bluetoothd`, `bluetoothaudiod`, the Bluetooth and Bluetooth audio daemons, as well as the frameworks `IOBluetooth` and `IOKit` and the Kernel extension / Bluetooth driver `IOBluetoothFamily`. It turns out that `IOBluetooth` is the deepest layer directly used by Apple software, before the more generic `IOKit` functions are called, and it contains the function that we are looking for: `_BluetoothHCISendRawACLData`. This is the only other *_BluetoothHCISend** function in `IOBluetooth` and the other one, being `_BluetoothHCISendRawCommand`, is a quite familiar one since it is already in use in our initial *InternalBlue* port to send HCI commands to the Bluetooth chip [10].

Interestingly, we found no _BluetoothHCISend function relating to SCO, which is a protocol for real-time narrowband audio.*

Of course, disassemblers are good at finding strings but struggle to retrieve parameter types which can often only be guessed. While stepping through the binary with a debugger would be one option, we chose a manual approach, especially since we already have the signature of the related functions `_BluetoothHCISendRawCommand` and `_BluetoothHCIDispatchUserClientRoutine`. One main difference to `_BluetoothHCISendRawCommand` is that the function for sending ACL has an additional parameter.

Listing 4.9 shows the function signatures already known from related work.

Listing 4.9: Private `IOBluetooth` API function signatures

```
int BluetoothHCISendRawCommand(uint32_t request,
                               void *commandData,
                               size_t commandSize);

int BluetoothHCIDispatchUserClientRoutine(struct
    IOBluetoothHCIDispatchParams *arguments,
                               unsigned char *returnValue,
                               size_t *returnValueSize);
```

Listing 4.10 shows the decompiled HCI and ACL functions contributed by this work.

Listing 4.10: Reversed private `IOBluetooth` API functions

```
int _BluetoothHCISendRawCommand(int arg0, int arg1, int arg2
    ) {
    memset(var_90, 0x0, 0x74);
    if ((arg1 != 0x0) && (arg2 > 0x0)) {
            result = _BluetoothHCIDispatchUserClientRoutine(
                arg0, 0x0, 0x0);
    }
    else {
            result = 0xe00002c2;
    }
    return result;
}

int _BluetoothHCISendRawACLData(int arg0, int arg1, int arg2
    , int arg3) {
    var_16 = arg2;
    var_17 = arg3;
    memset(var_90, 0x0, 0x74);
    if ((arg0 != 0x0) && (arg1 > 0x0)) {
            result = _BluetoothHCIDispatchUserClientRoutine(
                arg0, var_94, 0x4);
    }
    else {
            result = 0xe00002c2;
    }
    return result;
}
```

By declaring these functions in an Objective-C header file, they can be called from custom code. However, the parameter types need to be guessed correctly.

After some testing and checking log output via *PacketLogger* and *Wireshark*, we found that the order of the three common parameters, `request`, `commandData` and `commandSize` was not consistent across these related functions.

For example, finding what we thought was the `commandSize` parameter in the data portion of the packet or getting error messages like

"0x04 is not a valid device handle" helped a lot to understand what kind of parameter is expected at each position.

So, the signature of `_BluetoothHCISendRawACLData` turns out to be:

Listing 4.11: `BluetoothHCISendRawACLData` function signature

```
int BluetoothHCISendRawACLData(void *commandData,
                                size_t commandSize,
                                uint32_t handle,
                                uint32_t request);
```

The additional parameter in this function is a handle to a currently connected Bluetooth device. There are multiple ways to retrieve it and it is especially simple if the Medium Access Control (MAC) address of the device is known, but it is usually either `0x0b` or `0x0c`. In the case of Bluetooth Low Energy (BLE), Broadcom uses `0x40`.

When connecting to a new device while *InternalBlue* is running, the connection handle is displayed in the *InternalBlue* console and *PacketLogger* displays the handle of connected devices right at the top of its log.

HCI commands are always without handle since they can perform all kinds of operations, also ones that do not require a device to be connected. Commands that relate to a connected device can carry the device handle in the `commandData` section. However, the purpose of ACL is specifically to send data within an already established connection, so the `BluetoothHCISendRawACLData` function requires a connection handle.

The HCI command Read_AFH_Channel_Map is specific to a connection handle, which is passed as commandData. It retrieves the channel map and AFH mode for the given connection.

- Send HCI command:
  `(request, *commandData, commandSize)`
- Send ACL data:
  `(*commandData, commandSize, handle, request)`

4.5.2 *Sending ACL data*

`commandData` is a pointer to a memory location, so Listing 4.12 shows a full proof of concept for sending ACL data to a connected headset.

Listing 4.12: `BluetoothHCISendRawACLData` function call

```
BluetoothHCIRequestID request = 0;
BluetoothHCIRequestCreate(&request, 1000, nil, 0);

size_t commandSize = 4;
uint8 * data = malloc(commandSize);
data[0] = 0x00;
data[1] = 0x01;
data[2] = 0x03;
data[3] = 0x7c;

uint16_t handle = 0x000C;
```

```
BluetoothHCISendRawACLData(data, commandSize, handle,
    request);
```

Figure 4.3 is a screenshot from *PacketLogger* and confirms the results generated from Listing 4.12.

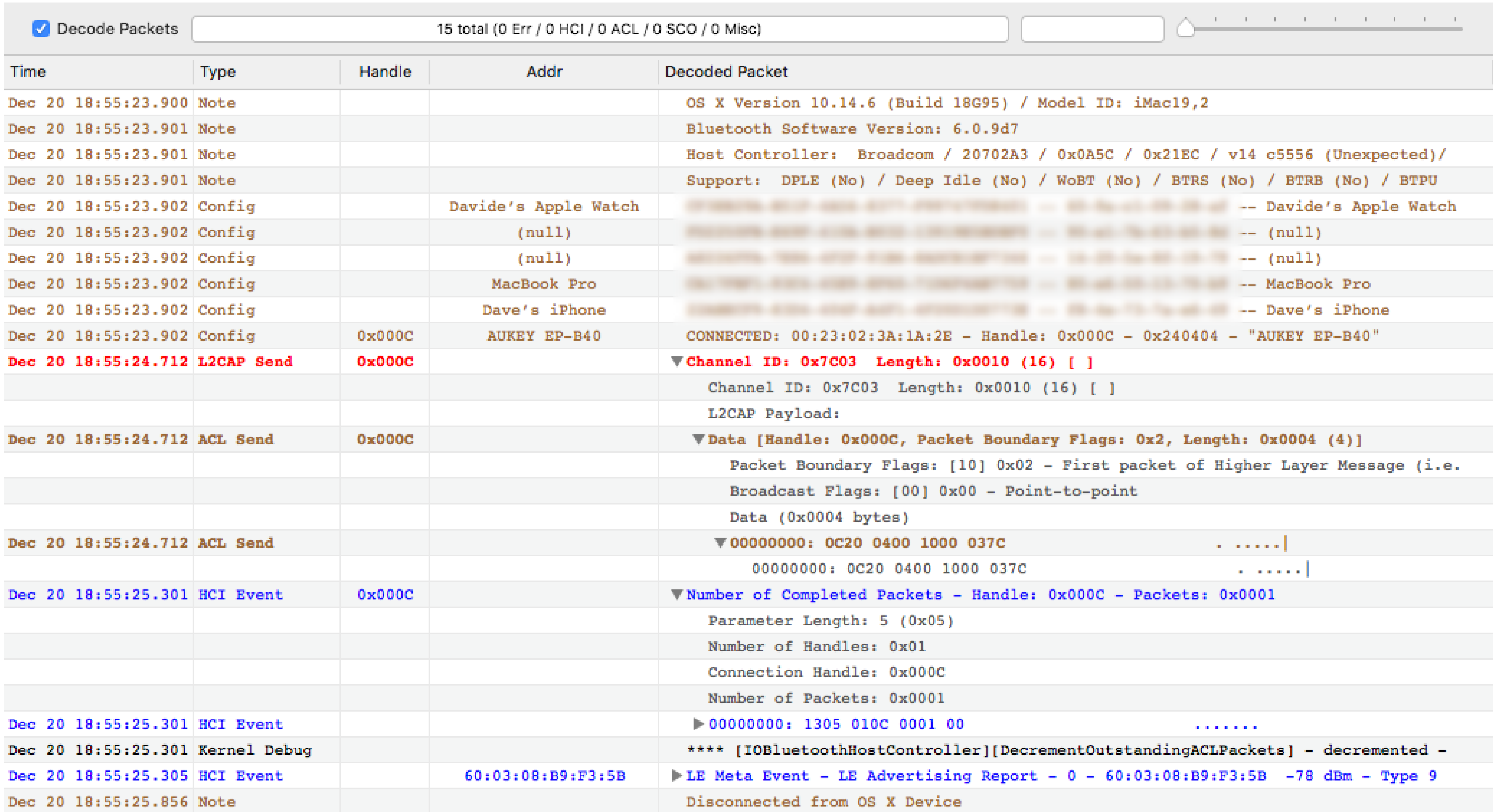

Figure 4.3: Proof of concept for sending custom ACL data to a Bluetooth client.

5 EVALUATION

The following chapter presents evaluations that have been made as part of this thesis. Section 5.1 shows how much concurrency affected the performance of data transfer between host and Bluetooth chip in the form of memory dumps. These tests helped to improve the performance of our *InternalBlue* port. Section 5.2 is a list of recent Mac models starting from 2015 containing Bluetooth chip model, Bluetooth firmware build date and version as well as the Bluetooth Link Manager Protocol (LMP) subversion for each computer.

5.1 PERFORMANCE USING DISPATCHQUEUES AND WITHOUT THEM

In macOS application programming, multithreading is usually done using `DispatchQueues` / Grand Central Dispatch (GCD). The concept was explained before, in Chapter 2 and its usage was shown in Chapter 4. Here is a small evaluation about how much it helped to speed up data transfers between Bluetooth chip and a host device.

Before using a `DispatchQueue` construct to send commands and data to the Host-Controller Interface (HCI) controller, the time to dump the Random Access Memory (RAM) of the Bluetooth chip was unviably long—almost 9 minutes. By using multithreading, that time was reduced to under a minute, usually around 30 seconds as reported in Table 5.1. The issue was that the same function of IOBluetoothExtended (IOBE) that waits for new commands (in this case the command to read the next 250 bytes) also sent the command to the HCI controller—on the same thread. Since it is a blocking function, IOBE cannot accept the next command until the result from the Bluetooth chip has returned. Using multithreading, it can dispatch that blocking function call to the background queue and it is instantly ready to receive the next command.

Since we use *Git* for version control and the whole project is fully open-source [7], the time improvement is easily reproducible using the following commit hashes: `f6fbe61` (before), `f632484` (after).

Commit Hash	Time
`f6fbe61`	8:29
	8:32
	8:40
`f632484`	0:33
	0:33
	0:34

Table 5.1: Time measurements with and without the usage of `DispatchQueues`.

5.2 LIST OF MAC MODELS, BLUETOOTH CHIPS AND FIRMWARES

To perform this analysis, we ironically used AirDrop to transfer our readRAM binary from an iPhone onto MacBooks and iMacs of various stores. Then we created screenshots of the logs and sent them back via AirDrop.

To quickly analyse as many Macs as possible, we built a little binary around the same framework that is also used in the macOS port of *InternalBlue* and hardcoded the `readRAM` command at the memory location `0x200400`. At that memory location, starting from around 2010, Broadcom Bluetooth chips have their firmware build date as a string. This analysis shows which Apple computers have very old firmwares, for that known exploits might exist and which ones have more recent firmwares. This information also helps to figure out which Bluetooth features are supported on each device.

Even though security updates can be delivered via macOS updates, the number of patches is very limited and thus old chips' patch slots can be quickly filled up. At that point, it has to be outweighed whether to overwrite older patches with new ones or not to patch the newly emerged issues.

The largest discrepancy between firmware build date and release date of the computer that carries the Bluetooth chip is the MacBook Pro 14,1 with the Broadcom 4350. The firmware (v127 c5602) was compiled in May 2013 and the MacBook was released in June 2017, more than 4 years later. The Mac with the most recent Bluetooth firmware relative to its release date is the MacBook Pro 15,4 that has the Broadcom 4377, a firmware built in February 2018 and was released in July 2019—less than a year later.

Duplicates of most listed models were tested and all data was coherent between devices with the same model identifier.

Usually, a year is totally fine since the Bluetooth chip must be manufactured before the product that uses it can really be developed, so that's an acceptable timeframe. But more than that could be due to too much old stock, even though in the meantime new Bluetooth chips and firmwares were developed.

Table 5.2 contains our results.

Model Identifier	Model Name	Release Date	Chip Model	Firmware Build Date	Firmware Version	LMP Subversion
MacBook Pro 16,1	16-inch	November 13, 2019	4364B3	May 9, 2018	v44 c4176	0x302C
MacBook Pro 15,4	13-inch, Two Thunderbolt 3 Ports	July 9, 2019	4377	February 28, 2018	v35 c111	0x2023
MacBook Air 8,2	13-inch	July 2019	4355	March 7, 2017	v47 c4906	0x102F
iMac 19,2	21.5-inch	March 19, 2019	4364B0	August 21, 2015	v66 c4392	0x1042
iMac 19,1	27-inch	March 19, 2019	4364B0	August 21, 2015	v67 c4398	0x1043
MacBook Pro 15,1	15-inch	July 12, 2018	4364B0	August 21, 2015	v67 c4399	0x2043
MacBook Pro 15,2	13-inch, Four Thunderbolt 3 Ports	July 12, 2018	4364B0	August 21, 2015	v67 c4399	0x2043
MacBook Pro 14,1	13-inch, Two Thunderbolt 3 Ports	June 5, 2017	4350	May 28, 2013	v127 c5602	0x617F
MacBook Air 7,2	13-inch	March 2015	20702B0	-	v150 c9318	0x4196

Table 5.2: List of Mac models, their Bluetooth chips and firmwares.

6 DISCUSSION

This chapter covers architectural design decisions for our *InternalBlue* macOS port in Section 6.1 as well as the discovery of an interesting and potentially vulnerable bug in macOS in Section 6.2.

6.1 COMMUNICATION ALTERNATIVES WITH IOBE FRAMEWORK

The communication between *InternalBlue* and the custom framework happens through socket communication with local User Datagram Protocol (UDP) servers. Before this solution was implemented, different alternatives were considered and tested.

6.1.1 *OpenStep Foundation events*

The `Foundation` framework that is essential when developing for macOS, iOS and watchOS, contains *"A notification dispatch mechanism that enables the broadcast of information to registered observers"* [4]. Objects can register to the default `NSNotificationCenter` as observers for specific system events or custom events with an associated function that is triggered when such an event is broadcast. `NSNotificationCenter` is officially supported by the *pyobjc* Python to Objective-C bridge so there are bindings which are accessible from Python. In our specific case, it is possible for the `macoscore.py` class to register as an observer of a custom notification.

Each process usually has its own "default NSNotificationCenter". This mechanism is not intended for interprocess communication.

In the first implementation of the macOS port, commands were sent to IOBluetoothExtended (IOBE) using direct function calls (see Section 6.1.2) and the results were returned via `NSNotificationCenter`, even though `Foundation` events would have been possible in both directions. When a result from the Bluetooth chip was received, the adapter framework broadcasted a message to the default `NSNotificationCenter` including the result data. Then, `macoscore` received it and could further process the event.

There are two main issues with this approach. The first one is that *InternalBlue* is completely designed around socket communication so after changing to the current model, with two local UDP servers, a lot of code could be removed and the `macoscore` simplified since more of the existing *InternalBlue* code could be reused and fewer methods had to be overridden. This also opens the ability to observe Bluetooth events using programs like *Wireshark*, which increases usability and is also available in Linux, Android and iOS versions of *InternalBlue*. The second issue with `NSNotificationCenter` is that more Python-

ObjC interaction was needed, which all has to go through the *pyobjc* bridge. Even though it is great that this bridge exists, it appears to be very scarcely documented and used, so there is little information to find about it online and hard to debug. `NSNotificationCenter` is also higher-level compared to sockets, so delays might be higher.

6.1.2 *Direct function calls*

As mentioned in Section 6.1.1, the very first setup was built such that in the `macoscore` commands were sent to the Bluetooth chip by directly calling the `IOBE` object's function to send commands to the Bluetooth chip. While it works fairly well and is the simplest way of communicating, we wanted to minimise the amount of Python-ObjC interaction and with the implementation of socket communication it could be reduced to the absolute minimum: only one line of code for the creation of the ObjC object `IOBE`, in which input and output socket numbers are passed. Function calls through *pyobjc* require a lot of caution to make sure that the parameters passed from Python match what is specified in the called function and it can get quite hard when trying to work with byte arrays.

6.1.3 *XPC messages*

XPC is an interprocess communication mechanism used in macOS and as documented in Chapter 2 also the way that `CoreBluetooth` communicates with `bluetoothd`. Since it is not usually meant to be used by userspace application developers, it is for the most part sparsely documented or not documented at all.

After finding out that this is how `CoreBluetooth` and `bluetoothd` communicate, the first idea that comes to mind is to use XPC ourselves to skip `CoreBluetooth` and its limitations. There are multiple issues with this approach, though. On one hand, the documentation is bad and going this route would require lots of trial and error as well as disassembly of `CoreBluetooth`. On the other hand, these are closed-source implementations without exposed Application Programming Interface (API), so Apple could change them at any time and thus break our implementation. Lastly, we found out how to skip `bluetoothd` and inject even deeper into the Bluetooth stack, on the last layer above the driver, thus using XPC and talking to the higher-level daemon was not useful in the context of this thesis. Still, there are projects like Bleno [17] that use this method.

6.2 MACOS BUG—READ MEMORY OUT OF BOUNDS

Due to a mistake in one of the first tests to read the Bluetooth chip's Random Access Memory (RAM) via Host-Controller Interface (HCI) commands, a macOS bug was found that allows reading system memory.

More specifically, one such command is built up of the following byte array: `{0x4D, 0xFC, 0xF0, 0x05, 0x00, 0x00, 0x00, 0xFB}`. The first two bytes indicate that it is an *HCI Read RAM* command (`0xfc4d`) and the third byte tells the HCI controller the length of the passed data, followed by command code and length parameter.

In the example array, the data section starts at byte 4 and ends at byte 8 (`0x05000000fb`). Clearly, our array only contains 5 bytes of data, but if we pass a `0xf0` (the number `240` in decimal) instead, the controller thinks that the command is 240 bytes long and reads whatever lies in the 240 bytes starting from the array's start address. The data can be inspected e.g. using the *PacketLogger.app* from Apple Developer Tools. This means that 235 bytes of data, in this case, are dumped, which might otherwise be impossible to read from an attacker without root rights. In fact, the code runs from any macOS user, even without root privileges.

We were not able to control the memory location to be leaked, but in one of many tests, it contained a path to the current user's login shell, which is very unlikely to be part of the executed binary.

Figure 6.1 is a screenshot of PacketLogger demonstrating the bug.

While this example performs a `readRAM` command, there is also a `writeRAM` command, which could leak memory from the host to the Bluetooth firmware. Combining these two, it might be possible to use memory of the Bluetooth chip as a side-channel.

We reported this bug to Apple together with minimal running code samples on August 19, 2019.

6.3 LIMITATIONS OF THIS WORK

OBFUSCATION This work has been possible due to the lack of obfuscation in Apple's binaries. The decompiler was able to retrieve all function names which made it straightforward to find the functions we were looking for through a string search. If in future versions of macOS, Apple strips function names, it will be hard to replicate this type of analysis.

API CHANGES A limitation of our code, exploiting a private and low-level API, may affect its long-term usability in case the API changes. If however, the function names will not be obfuscated again, the methodology applied in this work can be reused.

MULTITHREADING In a Python-independent implementation of memory dumping, we found out that events might arrive out of order. Since on macOS, these events don't have a time signature or other sequence identification methods, multithreading this task will lead

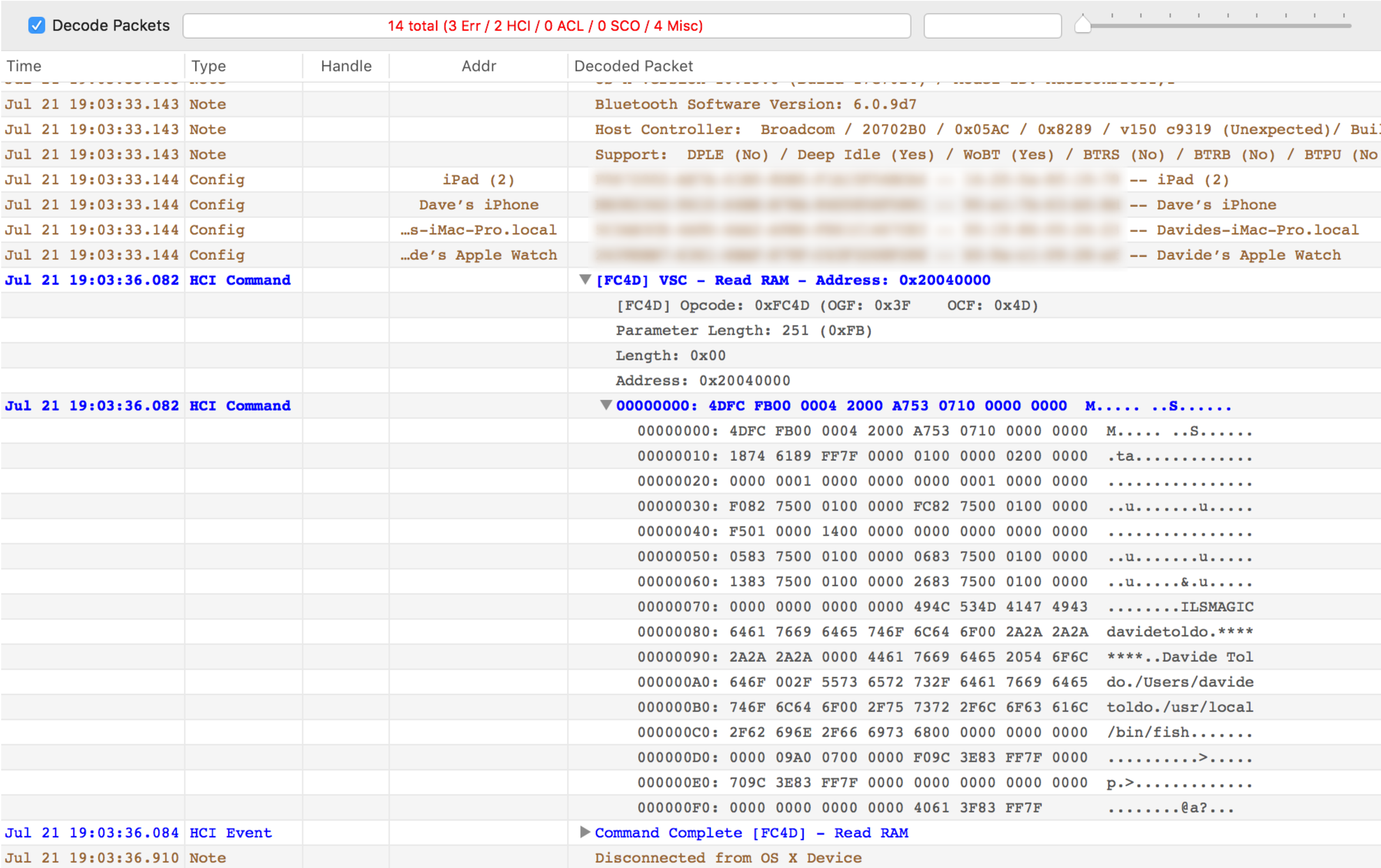

Figure 6.1: macOS bug allows to read memory out of bounds.

to unusable data. As the Bluetooth chip is not multithreaded, such behaviour was surprising for us. However, by manually reducing the HCI command sending rate we were able to create valid firmware dumps.

PYTHON Currently, *InternalBlue* is implemented in Python 2.7 which will not be maintained past 2020. Therefore, *InternalBlue* needs to be ported to Python 3 which is currently under development. However, since most of the work implemented here was done in Objective-C we do not foresee any limitations through the port.

HCI EVENT BYTE ORDER In this work (Table 4.1), we only found three HCI events with wrong byte order, delivered by the Bluetooth driver. However, there could be more. Either through a systematic manual approach or by decompiling the driver, it would be possible to have a full overview of all HCI events.

7 CONCLUSIONS

We unveiled the macOS Bluetooth stack and documented each layer between userspace applications and the chip. For better understanding of the implementation, we explained how frameworks and Kernel Extensions (Kexts) work and presented the relevant ones for Bluetooth. We showed how Apple uses a private API of their `IOBluetooth` framework to communicate using the Host-Controller Interface (HCI) and Asynchronous Connection-Less (ACL) protocols from a relatively high layer and how to use the functions ourselves. We built a custom framework that uses these private functions and runs a local User Datagram Protocol (UDP) server to communicate with the open-source Bluetooth experimentation suite *InternalBlue*. This way, we have created a fully functional macOS port and are now able to perform Bluetooth security research on macOS. While this is not part of the *InternalBlue* port yet, we provided a working proof of concept code example for sending ACL data to connected Bluetooth devices. A key factor in making the macOS *InternalBlue* port practically useful was to improve its performance since in the beginning, some tasks ran very slow. We resolved the issue using Grand Central Dispatch (GCD) and presented measurements of the ~10x time improvement. The ability to send arbitrary HCI commands was also crucial for an analysis of recent Mac models, where we documented for each tested Mac computer which Bluetooth chip it contains, when its firmware was built, which firmware version it is running and what its Link Manager Protocol (LMP) subversion is. This information was not accessible to date and helps identify Macs having or lacking specific vulnerabilities or features. Finally, we discussed a couple of alternatives for communicating between our custom framework and *InternalBlue*. Lastly, we documented a bug that we found during testing that allowed to read memory out of bounds and might be exploitable for side-channel attacks over Bluetooth.

A

APPENDIX

Listing A.1: User Datagram Protocol (UDP) server: startup (part 1/3)

```
private func startupServer() {
  let i = NWEndpoint.Port(self.inject as String)

  // Create UDP socket (SOCK_DGRAM)
  let sock_fd = socket(AF_INET, SOCK_DGRAM, 0)
  if sock_fd == -1 {
    perror("Failure: creating socket")
    exit(EXIT_FAILURE)
  }

  var sock_opt_on = Int32(1)
  setsockopt(
    sock_fd, SOL_SOCKET, SO_REUSEADDR, &sock_opt_on,
    socklen_t(MemoryLayout.size(ofValue: sock_opt_on)))

  var server_addr = sockaddr_in()
  let server_addr_size =
    socklen_t(MemoryLayout.size(ofValue: server_addr))
  server_addr.sin_len = UInt8(server_addr_size)

  // Set it to IPv4 Socket with specified port
  server_addr.sin_family = sa_family_t(AF_INET)
  server_addr.sin_port = UInt16(i!.rawValue).bigEndian

  // Bind socket with POSIX bind function
  let bind_server = withUnsafePointer(to: &server_addr) {
    Darwin.bind(sock_fd,
      UnsafeRawPointer($0).
        assumingMemoryBound(to: sockaddr.self),
    server_addr_size)
  }
  if bind_server == -1 {
    perror("Failure: binding port")
    exit(EXIT_FAILURE)
  }
```

Listing A.2: UDP server: main loop (part 2/3)

```
  // After binding successfully, run an infinite
  // loop on a background thread to receive
  // messages with hci commands
  DispatchQueue.global(qos: .background).async {
    while !self.exit_requested {
      // Prepare for receiving data
      var client_addr = sockaddr_storage()
      var client_addr_len = socklen_t(
        MemoryLayout.size(ofValue: client_addr))
```

```
      var receiveBuffer =
        [UInt8](repeating: 0, count: 1024)
      var bytesRead = 0

      // Receive data via recvfrom syscall
      bytesRead = withUnsafeMutablePointer(to:
        &client_addr) {
        $0.withMemoryRebound(to:
          sockaddr.self, capacity: 1) {

          recvfrom(sock_fd, &receiveBuffer,
            1024, 0, $0, &client_addr_len)
        }
      }

      if bytesRead == -1 {
        perror("Failure: error while reading")
        exit(EXIT_FAILURE)
      }
```

Listing A.3: UDP server: process command (part 3/3)

```
      // After reading the command, do all
      // further processing on a background
      // thread so the loop is executed
      // again and incoming commands
      // from the socket can be accepted
      DispatchQueue.global(qos: .background).async {
        // first 2 bytes are not needed
        var command = Array([UInt8](receiveBuffer).
          dropFirst(2))
        // extract the command length
        let length: UInt8 = receiveBuffer[1]

        // Send command to
        // Bluetooth HCI Controller
        // using custom framework
        // (method explained in 4.2)
        HCICommunicator.sendArbitraryCommand(
          &command, len: length)
      }
    }

    // Close sockets with generic system calls
    print("Exiting...")
    close(self.sock_fd)
    close(self.client_fd)
  }
}
```

BIBLIOGRAPHY

[1] Apple. *Kernel Extension Programming Topics*. https://developer.apple.com/library/archive/documentation/Darwin/Conceptual/KEXTConcept/KEXTConceptIntro/introduction.html. [Online; accessed 22-November-2019]. 2010.

[2] Apple. *System Integrity Protection Guide*. https://developer.apple.com/library/archive/documentation/Security/Conceptual/System_Integrity_Protection_Guide/KernelExtensions/KernelExtensions.html. [Online; accessed 15-December-2019]. 2016.

[3] Apple. *IOBluetooth*. https://developer.apple.com/documentation/iobluetooth. [Online; accessed 24-November-2019]. 2019.

[4] Apple. *NSNotificationCenter*. https://developer.apple.com/documentation/foundation/nsnotificationcenter. [Online; accessed 24-November-2019]. 2019.

[5] Keyvan Fatehi. *Writing your own kext kernel extension for mac os x*. http://keyvanfatehi.com/2011/06/11/Writing-your-own-kext-kernel-extension-for-mac-os-x/. [Online; accessed 22-November-2019]. 2011.

[6] Alexander Heinrich. *Die Magie hinter Handoff und Continuity*. https://macoun.de/video2019/gs7.php. [Online; accessed 14-December-2019]. 2019.

[7] Dennis Mantz. *Bluetooth experimentation framework for Broadcom and Cypress chips*. https://github.com/seemoo-lab/internalblue. [Online; accessed 14-December-2019]. 2019.

[8] Dennis Mantz, Jiska Classen, Matthias Schulz, and Matthias Hollick. "InternalBlue - Bluetooth Binary Patching and Experimentation Framework." In: *Proceedings of the 17th Annual International Conference on Mobile Systems, Applications, and Services - MobiSys '19* (2019). DOI: 10.1145/3307334.3326089. URL: http://dx.doi.org/10.1145/3307334.3326089.

[9] Alsey Coleman Miller. *HCI debugging tool for macOS*. https://github.com/colemancda/HCITool. [Online; accessed 14-December-2019]. 2018.

[10] Alsey Coleman Miller. *Hacking IOBluetooth*. http://colemancda.github.io/2018/03/25/Hacking-IOBluetooth. [Online; accessed 22-November-2019]. 2018.

[11] Alsey Coleman Miller. *Bluetooth Host Controller Interface Command Line Tool for for sending HCI commands on macOS and Linux*. https://github.com/MillerTechnologyPeru/hcitool. [Online; accessed 14-December-2019]. 2019.

[12] Peter Jay Salzman. *The Linux Kernel Module Programming Guide*. http://tldp.org/LDP/lkmpg/2.6/html/. [Online; accessed 22-November-2019]. 2007.

[13] Milan Stute. *An open Apple AirDrop implementation written in Python*. https://github.com/seemoo-lab/opendrop. [Online; accessed 14-December-2019]. 2019.

[14] Milan Stute. *An open Apple Wireless Direct Link (AWDL) implementation written in C*. https://github.com/seemoo-lab/owl. [Online; accessed 14-December-2019]. 2019.

[15] Milan Stute, David Kreitschmann, and Matthias Hollick. "One Billion Apples' Secret Sauce: Recipe for the Apple Wireless Direct Link Ad Hoc Protocol." In: *Proceedings of the 24th Annual International Conference on Mobile Computing and Networking*. MobiCom '18. New Delhi, India: ACM, 2018, pp. 529–543. ISBN: 978-1-4503-5903-0. DOI: 10.1145/3241539.3241566. URL: http://doi.acm.org/10.1145/3241539.3241566.

[16] InsanelyMac Wiki. *Kext (Kernel Extension)*. http://wiki.osx86project.org/wiki/index.php/Kext. [Online; accessed 22-November-2019]. 2017.

[17] noble. *bleno - A Node.js module for implementing BLE (Bluetooth Low Energy) peripherals*. https://github.com/noble/bleno. [Online; accessed 24-November-2019]. 2016.

ERKLÄRUNG ZUR ABSCHLUSSARBEIT

gemäß § 22 Abs. 7 und § 23 Abs. 7 APB TU Darmstadt

Hiermit versichere ich, Davide Toldo, die vorliegende Bachelor Thesis gemäß § 22 Abs. 7 APB der TU Darmstadt ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen. Mir ist bekannt, dass im Falle eines Plagiats (§ 38 Abs. 2 APB) ein Täuschungsversuch vorliegt, der dazu führt, dass die Arbeit mit 5,0 bewertet und damit ein Prüfungsversuch verbraucht wird. Abschlussarbeiten dürfen nur einmal wiederholt werden. Bei der abgegebenen Thesis stimmen die schriftliche und die zur Archivierung eingereichte elektronische Fassung gemäß § 23 Abs. 7 APB überein.

THESIS STATEMENT

pursuant to § 22 paragraph 7 and § 23 paragraph 7 of APB TU Darmstadt

I herewith formally declare that I, Davide Toldo, have written the submitted Bachelor Thesis independently pursuant to § 22 paragraph 7 of APB TU Darmstadt. I did not use any outside support except for the quoted literature and other sources mentioned in the paper. I clearly marked and separately listed all of the literature and all of the other sources which I employed when producing this academic work, either literally or in content. This thesis has not been handed in or published before in the same or similar form. I am aware, that in case of an attempt at deception based on plagiarism (§ 38 paragraph 2 APB), the thesis would be graded with 5.0 and counted as one failed examination attempt. The thesis may only be repeated once. In the submitted thesis the written copies and the electronic version for archiving are pursuant to § 23 paragraph 7 of APB identical in content.

Darmstadt, December 23, 2019

Davide Toldo