TECHNISCHE
UNIVERSITÄT
DARMSTADT

TOOTHPICKER: ENABLING OVER-THE-AIR AND IN-PROCESS FUZZING WITHIN APPLE'S BLUETOOTH ECOSYSTEM

DENNIS HEINZE

Master Thesis

March 31, 2020

Secure Mobile Networking Lab
Department of Computer Science

ToothPicker: Enabling Over-the-Air and In-Process Fuzzing Within Apple's Bluetooth Ecosystem
Master Thesis
SEEMOO-MSC-0170

Submitted by Dennis Heinze
Date of submission: March 31, 2020

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

Technische Universität Darmstadt
Department of Computer Science
Secure Mobile Networking Lab

ABSTRACT

Bluetooth is an integral part of the *Apple* ecosystem, which has gained even more significance with products like the *AirPods* and the *Apple Watch*. Many other features and technologies in the *Apple* ecosystem have been thoroughly researched with regards to security. Bluetooth, however, has only gotten little attention in public research. It has been primarily analyzed in terms of privacy. In general, mobile devices like *iPhones* and *Apple Watches* are an attractive target for attacks on short distance radio, as they are usually carried by their owners wherever they go. Additionally, flaws in Bluetooth protocols can lead to *zero-click* Remote Code Execution (RCE) attacks. Even worse, these attacks can easily be made worm-able and lead to infected devices attacking other devices.

The goal of this thesis is to explore and analyze *Apple*'s mobile Bluetooth stacks with a focus on their exposed protocols and their zero-click attack surface. We aim to shed some light into the Bluetooth stacks by reverse engineering and analyzing various Bluetooth-related components. Furthermore, we enable protocol-level security research by implementing *ToothPick*, a fuzzing framework capable of *over-the-air* and *in-process* fuzzing to target Bluetooth-related protocols, especially on *iOS*. Using the combined approach of manual analysis and fuzzing, we find and disclose various issues and vulnerabilities within the *iOS*, *macOS*, and *AirPods* Bluetooth stacks, including an issue that has existed since at least *iOS 5* (released in 2011).

ZUSAMMENFASSUNG

Die Bluetooth-Technologie ist ein zentraler Bestandteil des *Apple*-Ökosystems, welche in den letzten Jahren durch Produkte wie die *AirPods* oder die *Apple Watch* zunehmend an Bedeutung gewann. Viele der Features und Technologien von *Apple* wurden in den letzten Jahren im Hinblick auf ihre Sicherheit tiefgreifend analysiert. Bluetooth hingegen wurde in der öffentlichen Forschung hauptsächlich auf Privatsphäre untersucht. Grundsätzlich sind mobile Geräte wie *iPhones* und *Apple Watches* attraktive Ziele für Angriffe auf drahtlose Verbindungen, da die Besitzer der Geräte diese meistens mit sich führen. Schwachstellen in Bluetooth Protokollen können außerdem zu sogenannten *zero-click* RCE Angriffen führen. Im schlimmsten Fall können diese Angriffe auf Bluetooth auch in Form eines Wurms agieren, indem ein bereits infiziertes Gerät weitere Geräte angreift.

Das Ziel dieser Thesis ist, die mobilen Bluetooth Stacks von *Apple* zu untersuchen. Der Fokus liegt dabei auf den verfügbaren Bluetooth Protokollen und deren *zero-click* Angriffsflächen. Unter anderem geschieht dies durch Analyse und Reverse-Engineering von Bluetooth-Komponenten. Um die automatisierte Analyse von Bluetooth Protokollen zu ermöglichen, wird *ToothPicker*, ein *Over-the-Air* und *In-Process* Fuzzer entwickelt. Mit diesem können Protokolle auf dem *iOS*-Betriebssystem getestet werden. Durch eine Kombination von manueller Analyse und des Fuzzings werden außerdem einige Fehler und Sicherheitslücken identifiziert und an Apple gemeldet. Darunter befindet sich auch eine Sicherheitslücke, die mindestens seit der *iOS* Version 5 (von 2011) besteht.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LISTINGS

ACRONYMS

ACI	Apple Controller Interface
ACL	Asynchronous Connection-Less
AES	Advanced Encryption Standard
API	Application Programming Interface
ATT	Attribute Protocol
AVRCP	Audio Video Remote Control Profile
AWDL	Apple Wireless Direct Link
B-Frame	Basic Information Frame
BLE	Bluetooth Low Energy
BR	Basic Rate

CID	Channel ID
DoS	Denial of Service
ECB	Electronic Code Book
EDR	Enhanced Data Rate
GATT	Generic Attribute
HCI	Host Controller Interface
L2CAP	Logical Link Control and Adaptation Protocol
LEA	LE Audio
LEAP	LE Audio Protocol
LEAS	LE Audio Stream
LK	Link Key
MAP	Message Access Profile
MFi	Made for Apple
MSB	Most Significant Bit
MTU	Maximum Transmission Unit
PBAP	Phonebook Access Profile
PCIe	Peripheral Component Interconnect Express
PDU	Protocol Data Unit
PSM	Protocol/Service Multiplexer
RCE	Remote Code Execution
SCO	Synchronous Connection-Oriented
SD	Secure Digital
SDK	Software Development Kit
SDP	Service Discovery Protocol
SMP	Security Manager Protocol
SSP	Secure Simple Pairing
TCP	Transmission Control Protocol
TLV	Type-Length-Value

UART	Universal Asynchronous Receiver Transmitter
USB	Universal Serial Bus

1 INTRODUCTION

Apple's largely proprietary ecosystem has been subject to a vast amount of security research in the past few years. However, their Bluetooth stack experienced only little attention in public security research. Given their recently introduced Bluetooth accessories such as the *Apple Watch* and the *AirPods*, as well as recent features like *Handoff* or *Find My*, Bluetooth has gained significantly in relevance in the *Apple* ecosystem. Turning off Bluetooth implies drawbacks for the user, as many features depend on it to be enabled. However, devices with Bluetooth enabled are generally always connectable. Even when a device is set to be non-discoverable, the knowledge of the device's Bluetooth address suffices to initiate a Classic Bluetooth connection. In the *Apple* ecosystem, devices constantly emit advertising frames over Bluetooth Low Energy (BLE)[4]. While these are transmitted with a randomized device address, this address can still be used to connect to the respective device without any other prior knowledge. Additionally, the Generic Attribute (GATT) services that are, in accordance to the Bluetooth specification, available over this randomized BLE address, expose even more information about the device, such as the hardware model. Moreover, there are plenty of protocols exposed that do not require a pairing prior to communication. Bluetooth, in its nature, poses a large attack surface and potential target for zero-click Remote Code Execution (RCE) vulnerabilities. *Apple* recently announced a security bounty that also covers vulnerabilities on wireless channels, such as a $50.000 bug bounty for wireless chip RCEs and a $200.000 bug bounty for wireless RCE on the host [20].

Given the large attack surface and potential impact, it is crucial to properly assess the exposed applications and protocols. *Apple* only provides little public documentation regarding their proprietary Bluetooth protocols. In this thesis, we aim to uncover *Apple*'s Bluetooth stacks, with a special focus on the mobile ones, and lay the groundwork for future security research on this topic. We expose a variety of proprietary Bluetooth protocols and implementation details and implement *ToothPicker*, a fuzzing framework to target Bluetooth protocols.

1.1 CONTRIBUTION

The goal of this thesis is to explore and analyze *Apple*'s mobile Bluetooth stacks with a focus on their exposed protocols and their zero-click attack surface. The contributions of this thesis are as follows:

- We analyze and document the architecture and implementation of three of *Apple*'s Bluetooth stacks (*macOS*, *iOS*, and *RTKit*) with a primary focus on the mobile stacks (*iOS* and *RTKit*).
- We evaluate the zero-click attack surface of these stacks by means of protocols that are exposed prior to pairing.

- We implement *ToothPicker*, a fuzzing framework capable of over-the-air and in-process fuzzing. The over-the-air fuzzer can be used to test different Bluetooth stacks. The more in-process fuzzer can be used to fuzz Bluetooth protocol handlers in the Bluetooth daemon on *iOS*.
- We use these fuzzers to test a subset of the exposed Bluetooth protocols.
- We identify and responsibly disclose multiple vulnerabilities to *Apple*.

During the work on this thesis, we also wrote a paper on the *MagicPairing* protocol, which is currently under submission for the *ACM WiSec 2020* conference. Thus, it overlaps with Section 3.1, Section 5.2, and Chapter 6.

1.2 OUTLINE

First, we begin by laying the foundation of this thesis in Chapter 2. Next, the proprietary *Apple* Bluetooth ecosystem is documented in Chapter 3, along with an overview of their different Bluetooth stack implementations and their exposed protocols. In Chapter 4, we document the setup and methodology of our manual testing efforts, including an insight into the *AirPods* firmware and hardware. Following that, we present the concept and implementation of *ToothPicker*, our fuzzing framework consisting of an over-the-air fuzzer and an in-process fuzzer targeting Bluetooth protocols on *iOS*. We target a subset of the uncovered protocols and analyze the findings of our fuzzer. *MagicPairing*, one of the exposed Bluetooth protocols, is reverse-engineered and documented in detail in Chapter 6. Afterwards, the results are discussed in Chapter 7. We conclude the thesis in Chapter 8.

2 BACKGROUND AND RELATED WORK

Before getting into *Apple*'s Bluetooth stacks and their protocols, we first introduce the basics of Bluetooth that are required to understand some of the later sections. First, the Bluetooth technology is introduced in Section 2.1, then we briefly describe one of the higher level protocols in Bluetooth, the Logical Link Control and Adaptation Protocol (L2CAP) (Section 2.1.1), followed by a description of the Host Controller Interface (HCI). We conclude this chapter by presenting recent related work in the area of Bluetooth and the *Apple* ecosystem in Section 2.2.

2.1 BLUETOOTH

Bluetooth is a wireless communication technology used for short-range communication. It is mainly used for wireless communication in mobile devices. According to the *Bluetooth Market Report 2019* [23], there were around 3.8 billion devices shipped in 2018 that support the Bluetooth technology. Main applications for the technology are, amongst others, audio streaming, data transfer, and location services [23].

Bluetooth currently offers two different variants. Bluetooth *Basic Rate (BR)*, with the option for an *Enhanced Data Rate (EDR)*, and *Bluetooth Low Energy (BLE)*, with the latter one focussing on low power consumption. Bluetooth BR/EDR is often referred to as *Classic Bluetooth*. Both variants operate on the 2.4 GHz frequency band. To provide interoperability with other radio signals in that band, Bluetooth uses *frequency hopping*. In a typical Bluetooth setup, there are two components, a *host* and a *controller*. They communicate using the HCI. An overview of a generic Bluetooth protocol stack is shown in Figure 1. This serves as a reference for the *Apple* Bluetooth stacks shown in Section 3.1. Bluetooth offers various logical transports that are used to exchange data between devices, such as the Asynchronous Connection-Less (ACL) and Synchronous Connection-Oriented (SCO) protocols. For application layer protocols, the Bluetooth standard introduces the L2CAP, which is built on ACL. In the following, L2CAP will be introduced briefly.

2.1.1 *Logical Link Control and Adaptation Protocol (L2CAP)*

The L2CAP is an application-layer protocol that can be used for both connection-less and connection-oriented communication. It is built on top of ACL data frames and can be used with both Bluetooth BR/EDR and BLE. Features of L2CAP include protocol multiplexing, flow control, and segmentation and reassembly of data frames.

To support multiple upper-layer protocols over L2CAP, it introduces the concept of *channels*. There can be multiple channels for one Bluetooth connection, where each of the channels corresponds to a different upper-layer protocol. These channels are referred to by their Channel ID (CID). The CIDs between `0x0001` and `0x003F` are reserved for *Fixed*

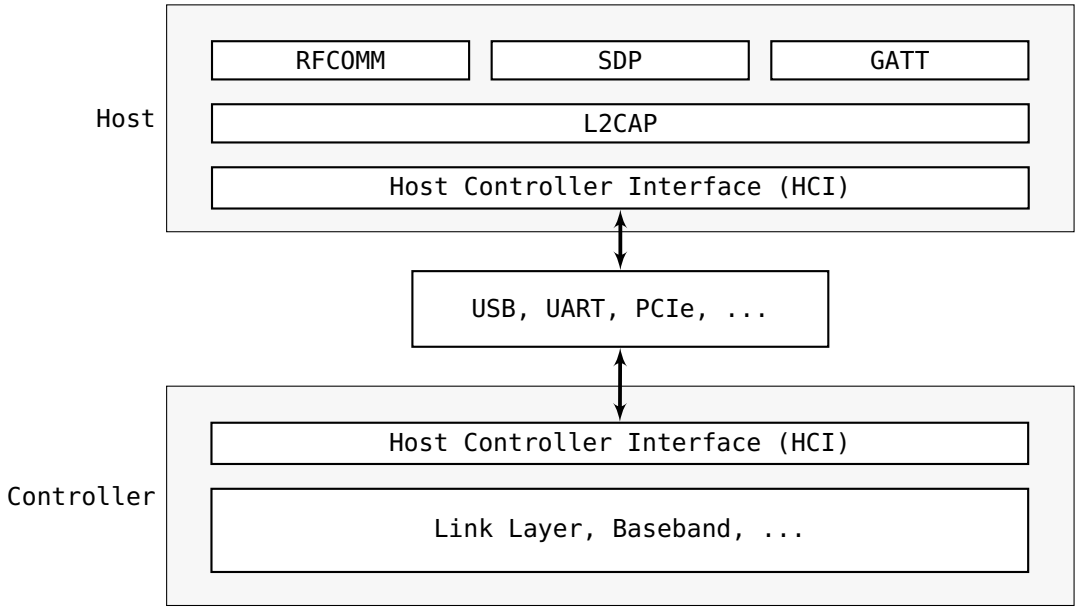


Figure 1: Overview of a Generic Bluetooth Stack

Channels. An example of a *Fixed Channel* is the *Signal Channel*, which is referred to by CID `0x0001`. This channel is responsible for managing dynamic L2CAP channels. When a new channel is to be opened, a *Create Channel* request has to be sent to the *Signal Channel*. There are other fixed channels that are specified in the Bluetooth standard, such as the CID `0x0004` for the Attribute Protocol (ATT) in BLE, or the CID `0x0002` for connection-less data [37]. According to the Bluetooth specification, CID `0x0000` is reserved and cannot be used. The remaining CIDs are dynamically allocated when opening a channel for any specified application-layer protocol. The Protocol Data Unit (PDU) of L2CAP payload data for connection-oriented channels—also called *Basic Information Frame (B-Frame)*—is rather simple. It consists of a length field, a CID field, and the payload data. The structure is shown below in Figure 2.

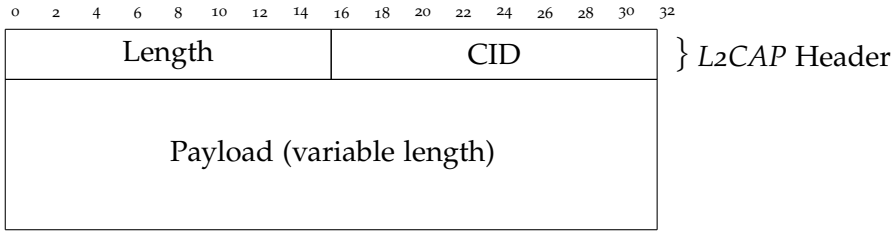


Figure 2: L2CAP B-Frame

2.1.2 *Host Controller Interface (HCI)*

The HCI is the interface that is used for communication between the host processor and the Bluetooth chip. It is independent of the underlying transport and supports any num-

ber of layers in between. Common transports within the *Apple* ecosystem are Universal Asynchronous Receiver Transmitter (UART), Universal Serial Bus (USB), or Peripheral Component Interconnect Express (PCIe). However, the Bluetooth standard only specifies the UART, USB, Secure Digital (SD), and the Three-Wire UART transport layers [37]. The HCI is used to exchange commands and data between the host Bluetooth stack and the Bluetooth chip firmware. The stack can send *HCI commands* and asynchronously receive *HCI events*. An example for this are the *Create Connection Command* and *Connection Complete Event*. In its simplest form, a Classic Bluetooth connection can be created by sending a *Create Connection Command*. If the connection is created successfully, the host will receive the *Connection Complete Event*.

2.2 RELATED WORK

Recently, there has been some research into features in the *Apple* ecosystem that rely on BLE advertising, namely *Handoff* and *Continuity*. These are described in the following subsection. Afterwards, some of the research into *Apple*'s *iPad* Bluetooth accessories is shown. Furthermore, we briefly mention the *BlueBorne* research. Lastly, the Bluetooth research framework *InternalBlue* is presented, which builds the basis of some of the research that has been done during this thesis.

2.2.1 *Apple Continuity*

Apple's *Continuity* feature is an umbrella term for a variety of interconnectivity features used between their various devices and operating systems. These features include *Universal Clipboard*, *Auto Unlock*, *Text Message Forwarding*, and many more [32]. While most of the protocols used for *Continuity* are built on Apple Wireless Direct Link (AWDL), protocol discovery and set up are mainly implemented via BLE advertisements [14]. Within the realm of *Continuity*, there are different BLE advertisement types for different features. One of them, which will be referred to later in this thesis, is the *Proximity Pairing* advertisement. The *Proximity Pairing* advertisements are sent out by *AirPods* and other *Apple* Bluetooth earphones. They contain various information about the respective earphones and are used to trigger the pairing process between them and a host device [4]. More on this process is shown in Chapter 6.

2.2.2 *iPad Bluetooth Peripherals*

In 2018, Esser published a white paper [5] and a conference talk [6] about the first generation *Apple Pencil* and *Magic Keyboard*. The focus of this research was mainly on the hardware and the firmware of these devices. Esser found various issues, such as the lack of a firmware downgrade protection of the *Apple Pencil*. However, the Bluetooth communication of these devices was not subject to this research. Instead, it documents several aspects of the firmwares, such as their binary format and how to extract containing data. Corresponding tools have been released on GitHub [36].

2.2.3 *BlueBorne*

BlueBorne was a collection of vulnerabilities in various Bluetooth stacks that was published in 2017. While the main target of this research was the Android Bluetooth stack, the authors identified a Remote Code Execution (RCE) vulnerability in the *iOS* Bluetooth stack. This vulnerability existed in the proprietary *LEAP* protocol. In addition to that, the authors also discovered a second proprietary L2CAP-based protocol called *Piped Dreams*[1]. The authors did not investigate this protocol further but pointed out that the complexity of this protocol also qualifies it as a target for further security research [39].

2.2.4 *InternalBlue*

InternalBlue is a *Python* framework that can be used for researching and communicating with *Broadcom* Bluetooth chips. It offers a wide range of features, including patching the Bluetooth chip's RAM via the `ReadRAM` and `WriteRAM` HCI commands [31]. To enable communication with the Bluetooth chip, *InternalBlue* requires an *H4* socket to send ACL, SCO, and HCI data. The HCI UART transport, commonly known as *H4*, is a protocol wrapper for the aforementioned protocols, where the first byte indicates the type of the following data [37]. In addition to the framework, Mantz et al. provide an insight into *Broadcom* Bluetooth firmware.

1 This protocol has since been renamed to *Magnet* (see Section 3.4).

3

THE APPLE BLUETOOTH ECOSYSTEM

In this chapter, we provide an overview over *Apple*'s rather complex Bluetooth ecosystem. We start by describing the different stacks that *Apple* currently ships in their main, Bluetooth-compatible, products (Section 3.1). Afterwards, in Section 3.2, we dig deeper into the *iOS* Bluetooth stack. Following that, we give a short introduction into *Apple*'s real-time operating system framework *RTKit* in Section 3.3. At the end of this chapter, we provide an overview over proprietary and custom Bluetooth protocols we identified within the *Apple* ecosystem (Section 3.4). Afterwards, we classify their potential as a research target (Section 3.5).

3.1 THE ARCHITECTURE OF APPLE'S BLUETOOTH STACKS

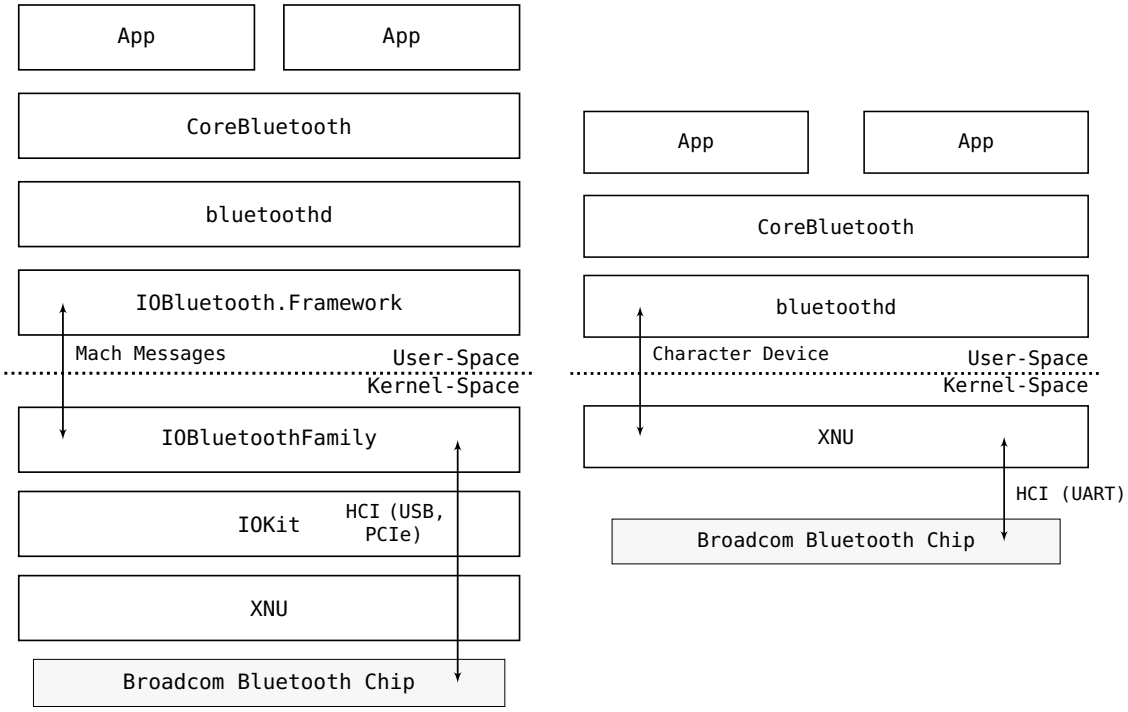
This section provides an overview over the Bluetooth stack in both *Apple*'s *iOS* and *macOS* operating systems. In this thesis we will refer to *Apple*'s mobile operating systems *tvOS*, *iPadOS*, and *watchOS* as *iOS*. All of them are based on *iOS* and are either renamed for marketing purposes, or due to other adaptions that are irrelevant to the Bluetooth stack. It is crucial to have an understanding of the architecture and implementation of the Bluetooth stacks to evaluate the attack surface and security implications of *Apple*'s Bluetooth ecosystems. In addition to *iOS* and *macOS*, a short overview over one of *Apple*'s Bluetooth peripherals, the *AirPods*, is provided. The reason for choosing *AirPods* is the lack of previous research into the security of these devices, as well as their high market share [25]. In general, the focus in this section is put on the mobile Bluetooth stacks, as the *macOS* Bluetooth stack has been thoroughly analyzed in previous work [42].

We begin by depicting the architecture and pointing out the differences between the *iOS* and *macOS* Bluetooth stacks. One of the major differences between the two operating systems' Bluetooth stacks is that the Bluetooth daemon *bluetoothd* plays a central role for all Bluetooth-related functionality on *iOS*, while it only has an administrative role on *macOS*. This difference is explained best by going into detail on how the operating systems communicate with the device's Bluetooth chip and how Bluetooth connections are created on the application layer.

On *macOS*, the operating system offers a driver built on *IOKit*[1], called *IOBluetoothFamily*, that implements the communication with the Bluetooth chip. In user-space, there exists a corresponding library, called *IOBluetooth*, that implements a client for the *IOBluetoothFamily* driver. A user space application can connect to an external Bluetooth device by using the *CoreBluetooth* library. In addition to that, developers might decide to use the *IOBluetooth* framework, which gives the developer more control than the *CoreBluetooth* framework. It implements undocumented but exported functions that can be called

1 *IOKit* is the driver subsystem in *Apple*'s operating systems. It provides both a kernel-mode framework to develop drivers, as well as a user-mode framework to communicate with these drivers [29].

to send Host Controller Interface (HCI) or Asynchronous Connection-Less (ACL) data directly to the Bluetooth chip [42]. However, *Apple* does not permit the usage of undocumented or private Application Programming Interfaces (APIs), as has recently been shown with apps using the *Electron* framework [33]. Thus, submitting apps to the *macOS AppStore* that use these undocumented *IOBluetooth* APIs will likely not be possible. When using the *IOBluetooth* framework, the Bluetooth daemon *bluetoothd* is not required. However, it still manages connections to system-wide Bluetooth devices, such as Bluetooth speakers, keyboards, or headphones and is involved when the *CoreBluetooth* framework is used. More details on the *macOS* Bluetooth stack can be found in [42].



(a) macOS (b) iOS (UART Bluetooth)

Figure 3: *iOS* and *macOS* Bluetooth Stack Differences

Due to the variety of devices supported by *iOS*, there are different ways the communication with the Bluetooth chip is implemented. A very common configuration seems to be a Bluetooth chip manufactured by Broadcom that is connected via Universal Asynchronous Receiver Transmitter (UART)[2]. As of the *iPhone* Xs, released in late 2018, the Broadcom chips are connected via Peripheral Component Interconnect Express (PCIe). The third configuration we observed was a Bluetooth chip manufactured by *Apple*, called *Marconi* (see Section 3.3.1). This configuration seems to be present on the *Apple Watch* starting from the third generation. Depending on the configuration, the Bluetooth daemon *bluetoothd* either connects to the character device representing the *UART* socket, the *IOService* exposed by the *Marconi* Bluetooth chip, or by using the `AppleConvergedTransport.dylib` library to connect to the chip via *PCIe*. The resulting stack architecture of this configuration is shown in Figure 4. More information on the different connection types is given later in Section 3.2.2.

2 We confirmed this for at least the *iPhone* 6, SE, 7, 8, X, and XR

After *bluetoothd* has connected to the Bluetooth chip, it starts to initialize the Bluetooth stack. Then, it offers Bluetooth functionality as an *XPC*[3] service. An app on *iOS* can use the *CoreBluetooth* framework to create Bluetooth Low Energy (BLE) connections and communicate with external Bluetooth peripherals. For this, the *CoreBluetooth* framework uses *bluetoothd*'s exposed *XPC* services. *iOS* does not allow apps to create and use Classic Bluetooth connections. Instead, it offers a higher-level application protocol called *External Accessory* that can only be used in combination with Made for Apple (MFi) certified Bluetooth devices. Figure 3 shows the differences between the two operating system's Bluetooth stack implementations.

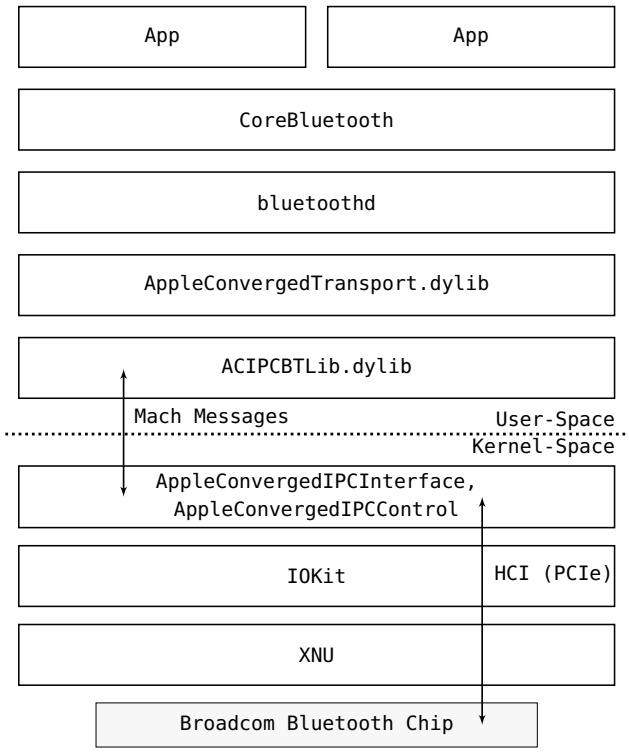


Figure 4: *iOS* Bluetooth Stack on Newer Devices

The third Bluetooth stack is the one used in the *AirPods*. While *Apple* has released multiple Bluetooth peripherals that are all based on the same real-time operating system framework *RTKit*, we decided to concentrate on the *AirPods* due to their high market share and lack of previous research. Based on the firmware analysis, we created an overview of the presumed architecture of the Bluetooth stack on the *AirPods* in Figure 5. The architecture is much simpler than the ones in the fully-fledged operating systems, as the devices supported by *RTKit* contain mostly single-purpose chips without any apps or the need for customizations. Therefore, privilege separation is not required. The *RTKit* operating system framework and the *Marconi* Bluetooth chip are explained in more detail in Section 3.3, and Section 3.3.1, respectively. Judging from the disassembled firmware, the *AirPods* Bluetooth stack and the *iOS* Bluetooth stack seem to share some of their code base. This can be observed in, for example, the reception handler function that parses incoming ACL data. The structure of the code parsing the different ACL fragments, as well as the

3 *XPC* is a low-level inter-process communication framework used across *Apple*'s operating systems [21].

fact that there is a fast path to the LE Audio (LEA) protocol (see Section 3.4) encourage this assumption. Some of the higher-level protocol handlers, such as the *MagicPairing* (see Chapter 6) reception handler, are implemented completely differently, though. However, by solely comparing the assembly or decompilation it is merely a guess that they share code. Details, such as the processor architecture, compiler optimizations, and compiler configurations influence the byte-code of the binaries. Therefore, the binaries alone are not a reliable source.

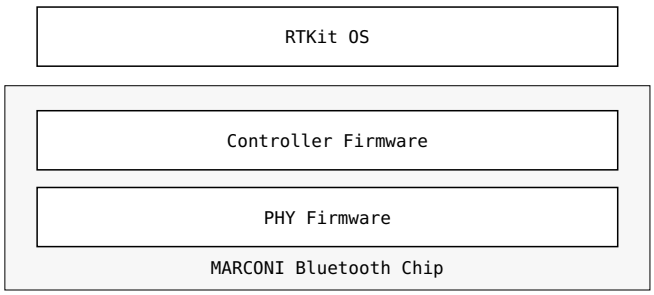


Figure 5: *RTKit* Bluetooth Stack

3.2 BLUETOOTH ON IOS

In this section, the *iOS* Bluetooth stack is examined in more detail. We begin by listing components and frameworks that are responsible for Bluetooth functionality on *iOS* in Section 3.2.1. Afterwards, we give an insight into the functionalities and responsibilities of *iOS*'s Bluetooth daemon *bluetoothd* in Section 3.2.2.

3.2.1 *Components and Frameworks*

On *iOS* devices with Bluetooth chips connected via UART[4], the operating system exports multiple character devices that are related to Bluetooth, which are listed below:

- `/dev/poweroff`
- `/dev/btwake`
- `/dev/{cu,tty,uart}.bluetooth`

The purpose of the `/dev/btpoweroff` character device is to reset the Bluetooth chip. Opening this socket will immediately power off the chip and turn it back on. This character device is not used directly by *bluetoothd*, but only by the *BlueTool* utility. The `/dev/btwake` device is used to wake the Bluetooth chip. If the Bluetooth chip is not required, e.g., because the host does not have an open Bluetooth connection or is currently not actively receiving BLE advertisements, it can be put to sleep to save energy. To wake the chip again, the host Bluetooth stack opens the character device `/dev/btwake`. As soon as the chip can be put back to sleep, the socket handle to the character device has to be closed again. The other devices (`cu., tty., and uart.`) represent the exposed UART

4 All *iPhones* before the *iPhone XS/11*, various *iPads*, and *Apple Watches* before the third generation.

socket to the Bluetooth chip. They can be used to communicate with the chip using the HCI. More information on this connection can be found in Section 4.1.

Newer *iPhones* (starting from the *iPhone XS/11*) have the Bluetooth chip connected via PCIe. These devices lack the aforementioned character devices. Instead, they use a library called `AppleConvergedTransport.dylib` to communicate with the Bluetooth chip over PCIe. For devices that use the *Apple Marconi* Bluetooth chip, the communication between the Bluetooth daemon and the chip is realized over an *IOKit*-based driver.

In addition to the character devices, libraries, and drivers, there are multiple daemons and tools that are concerned with Bluetooth on *iOS*:

BLUETOOTHD is *iOS*'s main Bluetooth daemon. It is responsible for communicating with the Bluetooth chip using the HCI and exposing several *XPC* services and a *Mach* service[5] to be used by other components in the operating system. As it is a vital Bluetooth component on *iOS*, more details about it are provided in Section 3.2.2.

BLUETOOL is both a utility and a service that is mainly responsible for initializing and updating the Bluetooth chip on *iOS*. It exposes an *XPC* service that is occasionally used by *bluetoothd*. Moreover, it can be used manually, as an interactive shell to communicate with the Bluetooth chip via HCI or Apple Controller Interface (ACI) (see Section 3.3.1).

BTLESERVER is a daemon that is mainly responsible for BLE connections. It handles *Apple*'s BLE peripherals, such as the *Siri Remote* or the *Apple Pencil*. Different to *bluetoothd*, it does not directly communicate with the Bluetooth chip. Instead, it communicates with *bluetoothd* via *XPC*, from which it receives relevant data.

BTAVRCP, BTPBAP, BTMAP are daemons that are responsible for the Bluetooth Audio Video Remote Control Profile (AVRCP), Phonebook Access Profile (PBAP), and Message Access Profile (MAP) profiles.

Lastly, there are various frameworks with different purposes with regards to Bluetooth functionality on *iOS*.

COREBLUETOOTH is *Apple*'s main Bluetooth framework for developers that want to incorporate Bluetooth functionality into their applications. It is available for both *iOS* and *macOS*, albeit being different on *macOS*. However, *CoreBluetooth* only allows BLE connections. As of *iOS* 13, developers need to specify why their app requires Bluetooth capabilities so that the user can decide whether they want to give the app the permission to use Bluetooth. If this information is not specified, the app will be terminated [22]. It is important to note that Bluetooth audio streaming does not require any of the Bluetooth entitlements.

MOBILEBLUETOOTH is a private framework on *iOS* that acts as a wrapper around *bluetoothd*'s `com.apple.server.bluetooth` *Mach* service. *Apple*'s daemons (e.g., *wifid*) use this framework to communicate with *bluetoothd*. It allows access to more privileged Bluetooth operations, such as retrieving the local device's Bluetooth address.

5 *Mach* services are the lowest layer of inter-process communication on *Apple*'s operating system based on *Mach Messages* and *Mach Ports* [18].

EXTERNAL ACCESSORY is used to connect to certified MFi devices. It enables an app to communicate with a device via Classic Bluetooth or the *iOS* device's Lightning connector. Before being able to use the framework and the corresponding *iAP2* protocol, the device needs to be certified and the developer needs to register at the *Apple* MFi program. The public documentation does not describe how *External Accessory* works on a protocol layer. The framework only provides the developer high-level programming interfaces while the rest, like connection creation and Bluetooth (or Lightning, respectively) transport is handled by the framework [19].

BLUETOOTHMANAGER is a private framework that exports various functions related to the pairing and management of Bluetooth peripherals.

BLUETOOTHAUDIO is a private framework that first appeared in *iOS* 13. It is mainly responsible for the newly introduced audio sharing feature which allows audio to be streamed to two pairs of *AirPods* simultaneously. This functionality is called *Wireless Splitter* internally.

3.2.2 *bluetoothd*

As *bluetoothd* is an important component in the *iOS* Bluetooth stack, this section describes its tasks and responsibilities. Most of the information presented here was gathered by analyzing and reverse engineering *bluetoothd* on *iOS* 12 and 13.

At startup, *bluetoothd* probes the device's *Device Tree*[6] for the implemented Bluetooth controller. As briefly mentioned in Section 3.1, *bluetoothd* supports different means of communication with the embedded Bluetooth chip:

- `H4/H4DS/H4BC/H5`
- `USB`
- `APPLEBT`
- `PCIe`

During startup, *bluetoothd* first checks whether there exists one of the following character devices:

- `/dev/cu.bluetooth`
- `/dev/uart.bluetooth`

The existence of either of these implies that the Bluetooth chip is connected via *UART* and, thus, uses any of the protocols *H4*, *H4DS*, *H4BC*, or *H5*. In this case, the operating system exposes the UART connection to the Bluetooth chip as a character device to user-space. The protocol we observed being used in this configuration is H4. However,

6 The *Device Tree* is a binary configuration file format that details the device specific hardware configuration. It differs for the various *iOS/macOS* devices and is used by the kernel to properly initialize the hardware [29].

we were only able to confirm this for a subset of *iOS* devices[7]. There might be other devices and configurations that use any of the other supported transport protocols.

If these character devices do not exist, the next device *bluetoothd* checks for is `marconi-bt`. This name represents *Apple*'s Bluetooth chip *Marconi*. More information on the *Marconi* Bluetooth chip can be found in Section 3.3.1. The transport protocol used to communicate to this chip is titled `applebt`. In contrast to the character device-based connection, the `applebt` transport is implemented as an *IOKit*-based kernel driver. The devices that support this protocol (such as the newer generation *Apple Watches*), have various Bluetooth related kernel drivers, such as the `com.apple.driver.MarconiBTFirmwareKext` driver, the `com.apple.drivers.AppleS7002BT` driver, and the `com.apple.driver.AppleBluetooth` driver, which are responsible for managing the Bluetooth chip. Using the *IOKit* framework, data can then be sent to and received from the Bluetooth chip.

The last device *bluetoothd* checks the *Device Tree* for is a device called `bluetooth`. While this entry also exists on all other devices with a non-*Marconi* Bluetooth chip, the absence of this entry suggests that the current device does not have a Bluetooth chip at all. In case both the `marconi-bt` and the `bluetooth` entries do not exist, *bluetoothd* exits with the error message `"No bluetooth on this device"`. The combination of the missing character devices but the presence of the `bluetooth` device in the *Device Tree* indicate a third configuration: A Bluetooth chip connected via PCIe. The communication with the Broadcom PCIe Bluetooth chip is implemented in a library called `AppleConvergedTransport.dylib`. The library exports functions to create a connection to the chip and read/write from/to the chip (e.g., `AppleConvergedTransportWrite`). The `AppleConvergedTransport.dylib` library is a wrapper around *iOS*'s native PCIe functions.

Once the Bluetooth chip setup is finished, *bluetoothd* continues by setting up the *XPC* and *Mach* handlers.

3.3 APPLE'S REAL-TIME OPERATING SYSTEM RTKIT

RTKit is *Apple*'s real-time operating system framework, which is used on multiple embedded controllers and in most of *Apple*'s Bluetooth peripherals. While *RTKit* is not publicly documented by *Apple*, it has been mentioned by other researchers [12, 26, 43]. Both the main firmware and the Bluetooth firmware on the *AirPods* contain references to the string `RTKit`, indicating that they are based on *RTKit*. The complete absence of any documentation and source code in combination with the lack of symbols make it difficult to thoroughly analyze the firmwares and the inner workings of *RTKit*. *RTKit* seems to be present in all of *Apple*'s recent Bluetooth devices. In Table 2 we show an overview over *Apple*'s Bluetooth peripherals, their current firmware versions, and the corresponding *RTKit* versions. The versions have been determined by extracting strings from the firmware binaries.

7 Namely, the *iPhone* 6, 7, 8, X, and XR.

Device Name	Firmware Version	RTKit Version
AirPods 1	6.8.8	*unknown*
AirPods 2	2C54	RTKit-1264.60.4
AirPods Pro	2C54	RTKit-1264.60.4
Siri Remote 1st Gen	*not using RTKit*	
Siri Remote 2nd Gen	118	RTKit_iOS-791
Apple Pencil 1st Gen	*not using RTKit*	
Apple Pencil 2nd Gen	0154.0093.0444.0060	RTKit_iOS-1063
Smart Keyboard	*not using RTKit*	
Smart Keyboard Folio	45	RTKit_iOS-941

Table 2: List of *Apple* Peripheral Devices Using *RTKit*

In the specific case of the *AirPods*, the various firmwares[8] seem to be all built on *RTKit*. Different to the newer generations of *AirPods*, the first generation's firmware does not evidently reveal the version. However, it contains strong hints that it is built on *RTKit*, such as the string `RTKSTACK`, which is used by *RTKit* to initialize the stack [29]. According to the strings in their firmwares, the first generations of the *Siri Remote*, the *Apple Pencil*, and the *Smart Keyboard* do not seem to use *RTKit*. However, this cannot be said with complete confidence as their firmwares are older than the ones of their successors. One possible scenario could be that they are built on a predecessor of *RTKit*. In addition to Bluetooth devices, *RTKit* powers a number of other devices and chips in the *Apple* ecosystem. An example is the AOP firmware which is included in most of *Apple*'s mobile devices (such as the *iPhone*, *Apple Watch*, and even the *AirPods*). Another firmware based on *RTKit* is the newer generation *Apple Watches*' Bluetooth firmware, for their custom Bluetooth chip *Marconi*. It turns out that this firmware is beneficial for analyzing other *RTKit* firmwares and *RTKit* in general. This is because the *Apple Watch* Bluetooth firmware, which is included in the *Apple Watch* firmware update is a *Mach-O*[9] file. This means that it contains information about the structure of the binary file, as well as a small number of symbols for function and variables names. Comparing functions between the Bluetooth firmware *Mach-O* file and the main firmware of the *AirPods* reveal that there is only a subset of functions that exists in both firmwares. These functions are mostly related to core functionality of the real-time operating system, such as thread scheduling, mutexes, and the heap management. This hints at the fact that *RTKit* is a framework, rather than a single operating system, that can be configured for the targeted embedded device's requirements.

In terms of strings that reveal version information, the *AirPods* firmwares are the most verbose. When searching for strings containing `rtkit`, all other firmwares mostly contain

8 The *AirPods* contain multiple processors for different purposes. Therefore, the firmware consists of multiple firmware binaries, e.g., the main firmware, the Bluetooth firmware, or the AOP firmware.

9 Mach-O is the binary format used in *Apple*'s Darwin operating systems. It includes information such as the architecture or sections of the binary [28].

the *RTKit* version string. The newest *AirPods Pro* firmware, for example, also reveals the following strings:

- `RTKitAudioFrameworkW2`,
- `RTKitOSPlatform-620.60.2~616`, and
- `RTKit2.2.Internal.sdk`.

The last one indicates that *Apple* has an internal Software Development Kit (SDK) that seems to be used to develop *RTKit* applications. It seems likely that *RTKit*'s versioning and naming system is not strictly defined yet. The other peripherals contain the term `iOS` in their version string, while the *AirPods* version string does not contain that term, but a longer version number separated by dots. Additionally, the found `RTKitOSPlatform` string unveils a different version number.

While we consolidate all these peripherals into a single Bluetooth stack, it is important to note that the firmware in all these devices is different. One of the reasons for this are the different requirements these peripherals have. The *Siri Remote*, the *Apple Pencil*, and the *Smart Keyboard* only use BLE, while the *AirPods* rely on both BLE and Classic Bluetooth. However, for the sake of this overview, we consolidate them into one category as they share the same code base, with at least the basic *RTKit* code being the same.

3.3.1 *Apple's Bluetooth Chip Marconi*

During the analysis of the *AirPods* and *Apple Watch* firmwares, we found that, different from their other devices, *Apple* does not seem to use Broadcom Bluetooth chips. In various components, we found the string `marconi` as reference to the implemented Bluetooth chips and firmwares. Additionally, during setup, *bluetoothd* on *iOS* checks whether there is a device called `marconi-bt`. The fact that the *Marconi* firmware on the *Apple Watch* also references *RTKit* leads to the assumption that this is a custom Bluetooth chip developed by *Apple*. Since the third generation of the *Apple Watch*, *Apple* implemented their own wireless chips into their watches (W2 for the *Apple Watch* 3 and W3 for the newer *Apple Watch* models 4 and 5). Like most of their Broadcom chips, these seem to be combo chips incorporating both WiFi and Bluetooth. When unpacking the firmware for the *Apple Watch* 3, one can find an `ACIBTFirmware` folder as well as an `aciwififirmware` folder containing the Bluetooth and the WiFi firmwares. In the case of the *Apple Watch* 3's *watchOS* version 5.3.5 firmware, the most recent one as of this writing, the *RTKit* version for both the Bluetooth and the WiFi firmware is `RTKit_iOS-973.250.56`. From the firmware's *Device Tree*, it can be observed that the Watch has a device called `marconi-bt` as well as a `marconi-wifi`.

The *AirPods* firmware itself does not contain any references to the string `marconi`. Additionally, there is no *Device Tree* for *AirPods*, so the name of the chip or device is not listed anywhere. However, we still assume that some variant of this *Marconi* chip is built into them. Firstly, the firmware bundle shipped for the *AirPods* includes two Bluetooth- related

```
<Command name="HCI_VS_Snoop_Switch" opcode="0xFC12" type="vc">
    <DocTag>external_w2</DocTag>
    <Desc>The snoop switch command is used by primary host to switch roles
        (primary/secondary) with secondary earbud.</Desc>
    <ExtDesc>
        <p>The snoop switch command is used by primary host to switch roles
            (primary/secondary) with secondary earbud.<br />
            Usage (must follow exact order of commands):<br />
            1) Primary and Secondary perform prioritizing to UTP link via
                HCI_VS_UTP_Prioritze_for_Bud_Swap command<br />
            2) Old primary and secondary perform ACL flow STOP for snoop link via
                HCI_VS_ACL_Flow command<br />
            3) Old primary perform switch snoop link via HCI_VS_Snoop_Switch command<br
                />
            4) New primary and secondary perform ACL flow GO for snoop link via
                HCI_VS_ACL_Flow command<br />
            5) New primary set new local BD_ADDR (to public bd_addr of old primary) via
                HCI_VS_Write_BD_ADDR command<br />
            6) New primary perform role switch on UTP link (in order to become master of
                the UTP link) via HCI_Switch_Role command<br />
            7) New secondary set new local BD_ADDR (to private bd_addr of old secondary)
                via HCI_VS_Write_BD_ADDR command<br />
            8) Primary and Secondary disable prioritizing to UTP link via
                HCI_VS_UTP_Prioritze_for_Bud_Swap command<br />
        </p>
        <p>Event(s) generated:
        </p>
        <p>When the Controller receives the snoop switch command, it shall send the
            Command Status event to the Host.<br />
            In addition, when the Link Manager determines the snoop switch is completed,
                the Controller, on both Controllers, shall send a VS Snoop Switch
                Complete event to each Host.
        </p>
    </ExtDesc>
    <Response event="HCI_Command_Status_Event" />
</Command>
```

Listing 3.1: Excerpt of the `ACI_HCILib` XML File Documenting *UTP*

files `acib` and `phyb`, which could be interpreted as *ACI Bluetooth*[10] and *PHY Bluetooth*, respectively. Both of these firmware files contain a reference to the *RTKit* version used by the main firmware.

What increased our confidence in the fact that the *AirPods* use the *Marconi* chip are various references in *PacketLogger*, which will be explained in more detail in Section 4.2.2. One of the older versions of this tool contains documentation, in the form of a file called `ACI_HCILib_2.xml`, about ACI commands that are used to synchronize and pair the two individual buds of the *AirPods*. Listing 3.1 shows an example of such a documentation snippet documenting the *UTP* protocol.

10 The *ACI* abbreviation appears multiple times within Bluetooth related *Apple* components. Older versions of *Apple*'s Bluetooth *PacketLogger*, for example, offer the option to connect to an *ACI* device. We assume this to be a variation of the Host Controller Interface (HCI) called *Apple Controller Interface*. However, we do not have any evidence backing up this hypothesis as it is only ever shown in abbreviated form. In the context of an *Apple*-developed Bluetooth chip, however, this assumption seems reasonable.

The *DocTag* value is *external_w2*, which seems to reference the W2 chip built into the *Apple* Watch 3. However, the description of the protocol mentions an `earbud`. We assume that this is a mistake and the documentation was actually meant to reference the W1 wireless chip that is built into the first generation *AirPods*. At the time this *PacketLogger* version was released, the only *AirPods* version on the market was the first generation containing the W1 chip. We assume that this is the reason why the H1 chip, which is built into the second generation *AirPods* and the *AirPods Pro*, is not documented in this file. In general, the `ACI_HCILib` file contains many references to HCI/ACI commands used by *Apple*'s wireless chips. Some of them also refer to a `marconi2` chip.

3.4 PROPRIETARY BLUETOOTH PROTOCOLS

We first have to define a set of targets that are to be analyzed to get a better understanding of the state of the security of *Apple*'s Bluetooth stacks. Due to the large number of protocols and the high complexity of the Bluetooth standard, it is not possible to cover the whole stack within the scope of this thesis. However, there are protocols and features that are more interesting than others. One interesting category of protocols are the ones that are available to an attacker prior to authentication. Such protocols have the potential to be vulnerable to so-called *zero-click* attacks. These are attacks that do not require any user interaction at all. Another category are proprietary protocols that *Apple* introduced for use in their own ecosystem. These are usually undocumented or not publicly named at all, yet they are available on hundreds of millions of *Apple* devices. Lastly, protocols that are supported and implemented on multiple of *Apple*'s platforms are interesting due to their high distribution. One category of protocols we explicitly leave out of scope are BLE advertisements. As mentioned in Chapter 2, there has been exhaustive work on protocols like *Handoff* and *Continuity* already [4, 14, 32].

Table 3 shows an overview of the proprietary protocols used in *Apple*'s Bluetooth ecosystem that we were able to uncover. Note that there might still be more protocols than mentioned here, which we did not come across during the timeframe of this thesis. In the following, the protocols and their purpose are briefly introduced. Due to the timeframe of this thesis, we were only able to analyze and reverse-engineer a subset of these protocols. For completeness, we also mention protocols we only analyzed partially or not at all.

3.4.1 *MagicPairing*

MagicPairing is a custom pairing protocol that aims to replace the pairing mechanisms defined in the Bluetooth specification [37]. It adds features such as seamless pairing of all devices connected to the same *iCloud* account. *MagicPairing* seems to be mainly used to pair *AirPods* to an *iCloud* account. This protocols has been thoroughly analyzed and is documented in Chapter 6.

Category	Protocol	iOS	macOS	RTKit
Fixed L2CAP Channels	*MagicPairing*	✓	✓	✓
	Magnet	✓	✓	-
	LEA{P,S}	✓	-	✓
	FastConnect Discovery	✓	✓	✓
L2CAP Channels	*External Accessory*	✓	✓	✓
	AAP	✓	✓	✓
	DoAP	✓	✓	✓
	Magnet Channels	✓	✓	-
	FastConnect	✓	✓	✓
	Apple Pencil GATT	✓	-	✓
Other	*BRO/UTP*	-	-	✓
	USB OOB Pairing	-	✓	-

Table 3: List of *Apple*'s Proprietary Bluetooth Related Protocols

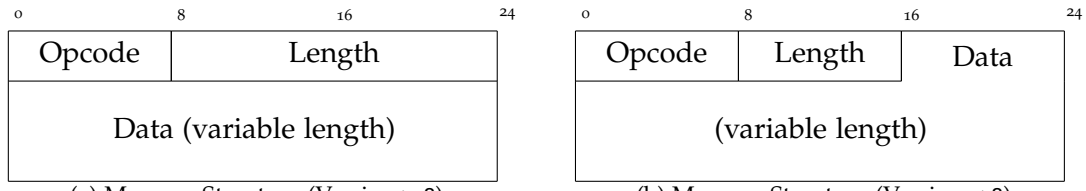
3.4.2 *Magnet*

Magnet is a protocol primarily used between an *iPhone* and an *Apple Watch*. On *macOS*, *bluetoothd* seems to refer to the protocol as *CBPipes* on occasion. Previously, this protocol has been referred to as *Piped Dreams* [39]. Its purpose seems to be similar to that of the Logical Link Control and Adaptation Protocol (L2CAP) Signal Channel. Furthermore, it can create and manage L2CAP channels. In addition to that, it offers synchronization features. *Magnet* seems to be tightly coupled with *Terminus*, which in turn seems to be a service responsible to communicate with the *Apple Watch* over multiple transports with different encryption methods. Another use of the *Magnet* protocol we have been able to observe is the *Wireless Audio Sync* feature of the *Apple TV* [40]. It allows a user to synchronize and correct timing offsets between the *Apple TV* and its connected speakers. For this purpose, the *Magnet* protocol is used to open another L2CAP channel which then handles the data exchange for the synchronization feature.

The *Magnet* message structure is shown in Figure 6. Depending on the version that was negotiated in the *Magnet* version message, the structure differs slightly. For a version newer than 8, the length field is two bytes instead of one. While observing the protocol in the wild, between an *iPhone* and *Apple Watch* as well as an *Apple TV*, we observed the protocol version 9 on *iOS* 13 and version 6 on *iOS* 12.

3.4.3 *LEA (LE Audio)*

LEA is a protocol that allows audio to be streamed over a BLE connection. This protocol consists of two subprotocols, one for the management of audio connections and the other for the actual stream. The protocols are called *LE Audio Protocol (LEAP)* and *LE Audio Stream (LEAS)*. While we did look into the implementation, we did not find a



(a) Message Structure (Version > 8) (b) Message Structure (Version < 8)

Figure 6: *Magnet* Message Structure

device that is using this protocol. According to sources citing the patent *Apple* registered for this protocol, it is aimed to be used for hearing aids, remote controls, and set top box devices [35]. *LEA* is a predecessor of the BLE audio protocol introduced in the new Bluetooth 5.2 standard [3] in 2020. It has existed at least since *iOS* 9 (released in 2015) [39]. Thus, *Apple* had the ability to stream audio over BLE at least five years before this technology was included in the Bluetooth standard.

During the analysis of the *LEAP* protocol, we found that there exists a *LEAP* version message that can be used to exchange version information between the sender and the receiver. The structure of this version message is shown in Figure 7. It contains the *LEAP* version, an encoded *Hardware Version* field, as well as an encoded *software version* field. While the hardware and software versions in these messages are encoded, they can be used to infer a device's operating system version. The *software version* for an *iPhone* running *iOS* 13.3 is, for example, `0xd30`. The algorithm that is used to encode the software version is shown in Listing 3.2. The version message, and the *LEAP* protocol in general, do not require an authenticated connection and are reachable via BLE. Thus, this serves as a valuable information leak for a potential attacker.

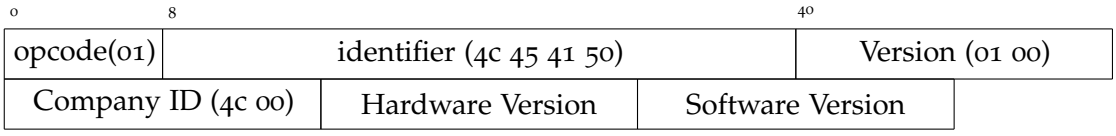


Figure 7: *LEAP* Version Message

```
# the iOS version as integer
i = 13
# the iOS version as double
d = 13.3
# a mutation of the version's double value
ver_double_mut = int((d*10.0) * 0x66666667) >> 0x20

version = (0 | i << 8 | ( int(d*10.0) + ((ver_double_mut >> 2) - (ver_double_mut >>
    0x1f)) * -10 ) * 0x10)
```

Listing 3.2: Algorithm to Derive *LEAP* Software Version Field

In Table 4 we give an overview over a few LEAP version messages captured from our test devices, along with their device and software versions.
In addition to the *LEAP* version message, the *Device Info* message of the *Magnet* protocol also contains a `vid` field which encodes the *iOS* version. However, we were not able

Device	iOS Version	Hardware Version Field	Version Field
iPhone 7	iOS 13.3	`7005`	`0D30`
iPhone 8	iOS 13.3	`710D`	`0D30`
iPhone 11	iOS 13.3	`730D`	`0D30`
iPhone 11 Pro	iOS 13.3.1	`7308`	`0D30`
iPad Mini 2 Retina	iOS 12.4.5	`6D0B`	`0C40`

Table 4: *LEAP* Version Message Leaks Captured in the Wild

to reliably get responses to *Device Info* requests sent to an *iPhone* or an *Apple Watch*. We were only able to observe the *Device Info* response in a *PacketLogger* trace where an authenticated connection between an *iPhone* and an *Apple Watch* was established. Therefore, we consider the version leak in the *Magnet* protocol as unproblematic.

3.4.3.1 *External Accessory*

The *External Accessory* protocol, also called *iAP2*, can be used as an application layer protocol to communicate with external hardware. The underlying frameworks abstract the actual transport layer. It supports connections over Classic Bluetooth or the Lightning connector. To use the protocol with an external device, it needs to be certified with the MFi program [19].

3.4.4 *AAP Protocol*

The *AAP Protocol* is a proprietary protocol by *Apple*, which is implemented on top of L2CAP. It is used for the communication between an *Apple* host device and the *AirPods*. The protocol offers multiple different services, which all revolve around configuring the *AirPods* and obtaining information and data from them. Examples are firmware updates, getting and setting tapping actions, or exchanging *MagicPairing* key material. The protocol is undocumented. However, analyzing the Bluetooth daemon on both *iOS* and *macOS*, as well as *Apple*'s Bluetooth *PacketLogger*, helps to reconstruct the protocol. The general structure of an *AAP* message is shown in Figure 8.

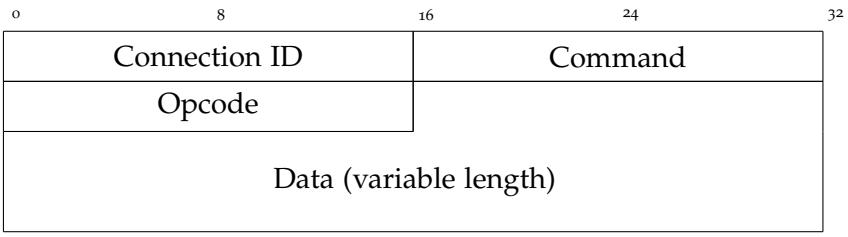


Figure 8: *AAP* Message Structure

3.4.5 *BRO and UTP*

BRO and *UTP*[11] are both protocols used between two individual *AirPods* buds. Their purpose includes synchronization and primary/secondary switching. Currently, the only source for information about these is found in *Apple*'s *PacketLogger* supplemental files (see Listing 3.1).

3.4.5.1 *DoAP*

DoAP is a protocol implemented on top of Generic Attribute (GATT). Its responsibilities include configuring and triggering *Siri*. Most of the relevant *DoAP* code resides within the *BTLEServer* binary. As it seems that the *DoAP* protocol is only available after pairing, it was not researched further in this thesis. However, being another proprietary protocol, it might be an attractive target for future work.

3.4.6 *Custom L2CAP Echo Responses*

L2CAP `Echo Requests` and `Echo Responses` have the purpose to implement a *ping*-like pattern between two Bluetooth peers. One party sends an `Echo Request` with a certain payload and the other party responds with an `Echo Response` which includes the same payload. Interestingly, *Apple* deviates from the Bluetooth standard here in two ways.

Firstly, if the `Echo Request` has a certain format, it will treat it as a *FastConnect Discovery Request* (the *FastConnect* protocol will be explained further in the following). Secondly, if the request payload is "Apple", it will respond with "Apple Computer Inc.". This behavior even differs between the various operating systems. On *iOS*, *bluetoothd* responds with "Apple Computer Inc.". On *macOS* this behavior is not present. Instead, it will respond with "Apple", according to the specification. The *AirPods* will respond with both "Apple" and "Apple Computer Inc.". This allows an attacker to disclose the device's operating system.

3.4.7 *FastConnect*

FastConnect is a protocol with the aim to improve the speed of connection setup. It simplifies configuration that would otherwise be exchanged through various Service Discovery Protocol (SDP) messages. Additionally, it removes the need to open dynamic L2CAP channels that have been used before. There are two parts of the *FastConnect* protocol, one that is used prior to pairing, and one that is used after the devices are paired.

The first part of *FastConnect*, the *Discovery* messages, are implemented on top of L2CAP `Echo Requests`, which is rather unusual, but simplifies the implementation efforts. There are two properties that indicate a `Discovery Request` as opposed to a normal `Echo Request`. The first indication is a payload length greater than 23. As soon as the L2CAP `Echo Request` payload is longer than 23 bytes, it is no longer interpreted as an `Echo`

11 We found no indication of what the acronyms *BRO* and *UTP* might refer to.

`Request` but as a *Fast Connect Discovery Request*. The second indication is the value `0x01` at byte position `6` in the L2CAP payload, which is required for the payload to be properly parsed. We are unsure why *Apple* chose this complicated approach instead of adding an additional opcode. The Bluetooth standard would be violated in either case.

The second part of the *FastConnect Protocol* is implemented over a dynamic L2CAP channel. This channel is created after sending a *FastConnect Discovery Request* followed by a successful pairing (e.g., via *MagicPairing*, which can be specified as a pairing `sink`). The Channel ID (CID) that is to be used for this protocol is specified in the *FastConnect Discovery Response*. The protocol then continues to exchange four messages, two for each peer: a *FastConnect Service Descriptor* followed by a *FastConnect Service Descriptor Response*. Afterwards a *FastConnect Service Configure* is followed by a *FastConnect Setup Complete*. The *FastConnect Service Descriptor* message contains a list of protocols the sender supports along with their configuration and allocated CID. The receiver acknowledges this list by sending a response containing the protocols it supports and its local CIDs. If required, the sender can then send further configuration options via the *Service Configure* message. The protocol concludes with the *Setup Complete* message, where the receiver acknowledges the received configuration. Afterwards, the negotiated protocol CIDs are opened and can be used by the peers.

As it is reachable via a fixed L2CAP channel, and thus prior to pairing, the discovery part of the *FastConnect* protocol introduces an additional attack surface. However, the attack surface is rather small as it only concerns two message types.

3.4.8 *USB Out-of-Band Pairing*

The *Apple Smart Keyboard* supports an out of band-like pairing mechanism. The keyboard first needs to be connected to the host device via USB. Then the Bluetooth Link Key (LK) is generated on the keyboard and sent to the host. The LK is transmitted through the HID configuration endpoint. Afterwards, the devices are automatically paired. While this is not a wireless protocol, we still want to mention this for completeness sake.

3.4.9 *Apple Pencil GATT*

During a dynamic analysis using *PacketLogger*, we found that the communication between the *Apple Pencil* and an *iPad* is implemented over GATT. Every movement of the pencil is transmitted as a GATT *Value Notification*. We were only able to test the first generation of the *Apple Pencil*. As the second generation only works with the newest *iPad* generations, the protocol might differ.

3.5 PROTOCOL PRIORITIZATION

In this section, we perform a prioritization of the protocols to derive which ones are interesting for further analysis. This prioritization is based on the following criteria:

OPERATING SYSTEM determines how many devices are affected or supported by the protocol. Can be any of the three: *iOS, macOS, RTKit*.

ACCESSIBILITY determines how much effort is required to communicate this protocol, e.g., does it require initialization, pairing or other preparation. In general, Fixed L2CAP Channel protocols have a very high accessibility, as they can be reached over an unauthenticated Bluetooth connection. Some protocols are reachable prior to authentication, but only unleash their full capabilities once the connection is authenticated and encrypted.

PROPRIETARY indicates that the protocol is not defined in the Bluetooth specification but instead developed by *Apple*.

KNOWLEDGE describes how much is known about the protocol. This is influenced by many factors. A protocol with a high knowledge, for example, is the *MagicPairing* protocol. For this protocol we have devices available that communicate using this protocol, can dissect traces of this protocol with *PacketLogger*, and have three different implementations that can be reverse-engineered to understand the protocol. A slightly different example is the *LEA{P,S}* protocol. We also have two implementations that can be reverse-engineered, *PacketLogger* can decode this protocol, but we do not have any devices that could be used for dynamic analysis or to even create a protocol trace to begin with.

The last column, *Target* determines whether a protocol was chosen for further analysis. We decided to choose *eight* protocols, with *five* of them being proprietary ones and *three* of them being very critical standard protocols. While there are other protocols from the Bluetooth specification that have a high accessibility and distribution within the operating systems, such as the *BLE Signal Channel*, or the *SecurityManager* protocols, we limited the chosen protocols to ACL, GATT, and the *Signal Channel*. Due to their low accessibility, which in most cases means an authenticated connection and complex protocol setup, we left out the dynamic L2CAP channel protocols.

Category	Protocol	iOS	macOS	RTKit	Accessibility	Proprietary	Knowledge	Target
Fixed L2CAP Channels	*MagicPairing*	✓	✓	✓	↑	✓	↑	✓
	GATT	✓	✓	(✓)	↑		↑	✓
	Signal Channel	✓	✓	✓	↑		↑	✓
	Magnet	✓	✓		-	✓	-	✓
	LEAP	✓		✓	-	✓	-	✓
	LEAS	✓		✓	-	✓	↓	✓
	FastConnect Discovery	✓	✓	✓	↑	✓	↑	✓
	DoAP	✓	✓	✓	?	✓	-	
	BLE Signal Channel	✓	✓	?	↑		↑	
	Connectionless Channel	✓	✓	?	↑		-	
	Classic Security Manager	✓	✓	?	↑		↑	
	BLE Security Manager	✓	✓	?	↑		↑	
L2CAP Channels	*SDP*	✓	✓	✓	↑		↑	✓
	AAP	✓	✓	✓	↓	✓	↑	
	External Accessory (iAP2)	✓	✓	✓	↓	✓	-	
	FastConnect	✓	✓	✓	↓	✓	-	
	Magnet Channels	✓	✓		↓	✓	↓	
Other	*ACL*	✓	✓	✓	↑		↑	✓
	BRO/UTP			✓	?	✓	↓	
	USB OOB Pairing		✓		-	✓	-	

Table 5: List of Bluetooth Protocol Targets

Accessibility describes how accessible the protocol is to an attacker. This includes protocol setup or authentication requirements of the protocol. There are three possible options indicated by arrows, *high*, *medium*, and *low*. The *Proprietary* column indicates whether the protocol is a proprietary protocol by *Apple*. *Knowledge* determines how much information about this protocol is available and how easliy it can be analyzed. *Targets* indicates that the protocol is targeted for further analysis. The brackets around the checkmars on *RTKit* indicate that this protocol is not available on all *RTKit* devices.

4 MANUAL TESTING

In this chapter, we describe the manual testing approach and tooling that was used to analyze some of the protocols in *Apple*'s Bluetooth stack. First, we describe how we extended *InternalBlue* to work on *iOS*. Next, we present our approach of dynamically and passively analyzing Bluetooth related components. We conclude this section with an overview of the specific testing that was done with regards to the *AirPods* and the *RTKit* real-time operating system.

4.1 INTERNALBLUE ON IOS

InternalBlue needs to connect to an `H4` socket to communicate with a Bluetooth chip. *iOS* provides exactly this type of socket through its Universal Asynchronous Receiver Transmitter (UART) Bluetooth character device. Therefore, the socket can be relayed to the host running *InternalBlue* using a proxy. We wrote a proxy daemon that forwards *iOS*'s Bluetooth socket via Transmission Control Protocol (TCP), which is then multiplexed over Universal Serial Bus (USB) using *usbmuxd*. More information about *usbmuxd* can be found in the following paragraph.

The proxy consists of two parts: the proxy daemon *internalblued*, as well as the settings bundle that is added to the *iOS* device's Settings application. The settings can be used to configure and restart the daemon. One configuration option, for example, is the port the proxy is listening on. A screenshot of the settings menu is shown in Figure 9. In addition to the user interface dependent approach, the daemon can be launched manually from the command line of the *iOS* device. The *iOS* device needs to be jailbroken[1] for *internalblued* to work. Both installing a *launch daemon* and accessing the Bluetooth chip are privileged operations on *iOS* that are not allowed for apps on the App Store.

In the following, some details on the implementation of the proxy daemon are provided. Note that this approach only applies to *iOS* devices where the Bluetooth chip is connected via UART. The newer models (i.e., *iPhone* Xs and 11) connected via Peripheral Component Interconnect Express (PCIe) need a different approach. In Section 7.2, we describe how this can be implemented. When the daemon first starts, i.e., after a restart of the operating system or after restarting the service, it sets up the proxy server by starting a new thread and listening on the configured TCP port. It will only connect to the local Bluetooth chip once a connection to the proxy is created. This ensures that the proxy can run alongside the operating system's Bluetooth stack. Usually, the character device only allows one active connection at a time. When a connection to the proxy is opened, the *iOS* Bluetooth stack should be disabled in the device's settings, preventing any interference

1 *Jailbreaking* refers to the act removing restrictions and sandboxing mechanisms on the *iOS* operating system. Usually, the goal is to install third-party software that would otherwise not be allowed on the *AppStore*. Additionally, *Jailbreaks* enable research and the development of research tooling for *iOS* [24].

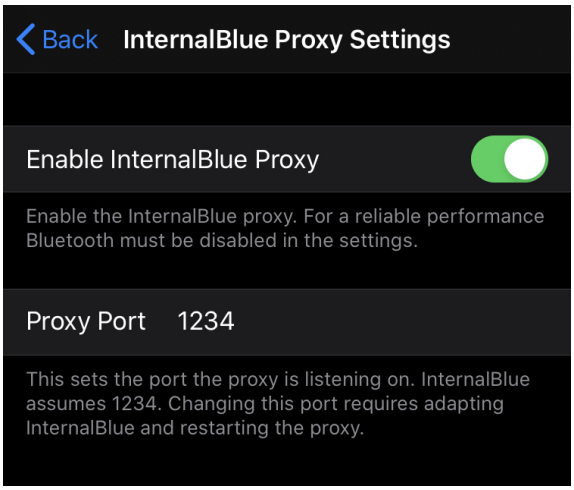


Figure 9: Screenshot of the *internalblued* Preferences

from *bluetoothd*. Afterwards, *internalblued* can safely connect to the Bluetooth chip. First, it needs to open the character device `/dev/btwake` to wake up the Bluetooth chip. Then, it enters the procedure to open the Bluetooth *H4* socket. In order to understand how a connection to the Bluetooth chip is established, *iOS*'s Bluetooth daemon *bluetoothd* as well the Bluetooth utility *BlueTool* have been reverse-engineered and the connection creation has been reimplemented. After *InternalBlue* has created a connection to the proxy, all data received at the proxy server is forwarded to the Bluetooth chip and all data received from the Bluetooth chip is forwarded to *InternalBlue*. The proxy server, by default, only listens to the local interface to not unnecessarily expose the Bluetooth socket of the testing device. *InternalBlue* can then create a connection to the proxy port over USB by leveraging *Apple*'s *usbmux* protocol. In short, *usbmux* is a protocol that allows TCP connections to be multiplexed over USB [44]. When an *iOS* device is connected to a computer via USB, a connection to the socket `/var/run/usbmuxd` is opened. After an initial handshake, that involves specifying the port that is to be tunneled from the *iOS* device, the tunneled port is available from the opened *usbmuxd* socket.

Lastly, we wrote an *iOS* core adapter for *InternalBlue* that handles the setup and configuration of the *iOS* devices connected to the computer via USB as well as the configuration of the *usbmuxd* service. As the data originating from the proxy occasionally arrives in bulks, we implemented simple parsing functionality as to split the incoming packets according to their length field. Then, these packets are forwarded to the usual logic of *InternalBlue*. Figure 10 shows the architecture of our *InternalBlue* proxy implementation for UART-based *iOS* devices.

In addition to adapting *InternalBlue* to *iOS*, we built a *testing* Bluetooth stack, which implements a very small subset of a Bluetooth stack on top of *InternalBlue*. The goal is to simplify writing proof-of-concepts and implementing an over-the-air fuzzer (presented in Section 5.1). While *InternalBlue* already allows creating connections and wraps several Host Controller Interface (HCI) commands in functions, it does not implement any logic that is, for example, tracking the connection and its handle. A Classic Bluetooth connection usually works as follows. The connection is created by sending a *Create*

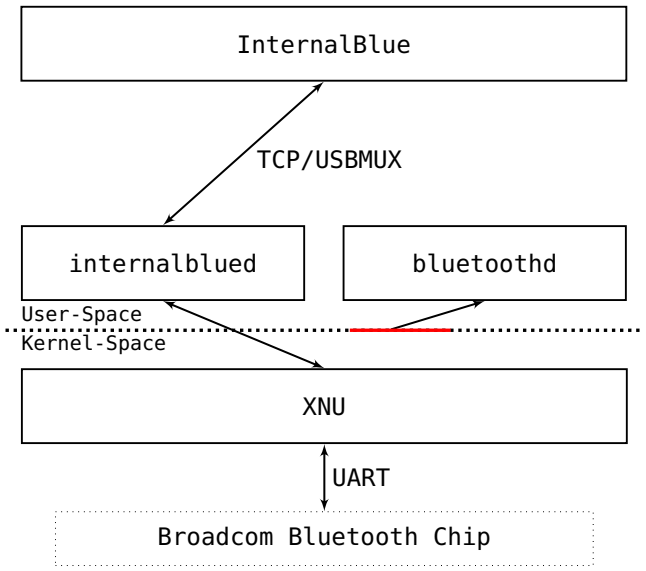


Figure 10: *InternalBlue iOS* Architecture (UART Bluetooth)

Connection HCI command with the targeted device's Bluetooth address as one of its arguments. If the connection was successful, an HCI event is sent to the Bluetooth stack along with the *connection handle* that was created for this particular connection. Every other command related to this connection, e.g., sending Asynchronous Connection-Less (ACL) data, requires this handle as argument. Once the connection is disconnected, a *Disconnection Complete* event is sent to the Bluetooth stack. We built the logic for connection handling as well as for sending, receiving, and parsing Logical Link Control and Adaptation Protocol (L2CAP) data.

4.2 STATIC AND DYNAMIC ANALYSIS

A large part of analyzing *Apple*'s Bluetooth implementations involves static and dynamic analysis, i.e., reverse-engineering the components that implement Bluetooth functionality as well as capturing and analyzing Bluetooth traffic. In this section, we provide details on how this analysis was done.

4.2.1 *Reverse-Engineering Bluetooth Related Components*

There are various components in the different operating systems that are related to Bluetooth functionality. Most of them provide a valuable source of information for understanding and analyzing Apple's Bluetooth stack. Interesting components include the Bluetooth daemons (*bluetoothd*), Bluetooth frameworks (e.g., *CoreBluetooth* or *MobileBluetooth*), the *AirPods* firmware, and kernel drivers responsible for handling the Bluetooth chip. Analyzing these components turns out to be of various difficulty. Factors that influence the difficulty of reverse-engineering these components are:

- The programming language used,
- the binary format of the component, and

- the presence or absence of symbols and function names.

Interesting to note is that *bluetoothd* is implemented differently on *macOS* and *iOS*. On *macOS*, it is mainly built on Objective-C, while the *iOS* implementation relies mostly on C and C++. On *iOS*, there are very few symbols available. On *macOS*, due to the nature of Objective-C, many method names are still present in the binary, which makes reverse-engineering the *macOS* version of *bluetoothd* much easier. Additionally, it is less complex than the *iOS* version, as it lacks the management of the Bluetooth chip and other functionality that is deferred to the Bluetooth kernel driver (see Chapter 3). Having two different implementations of *bluetoothd* is also helpful for protocols that are implemented in both versions.

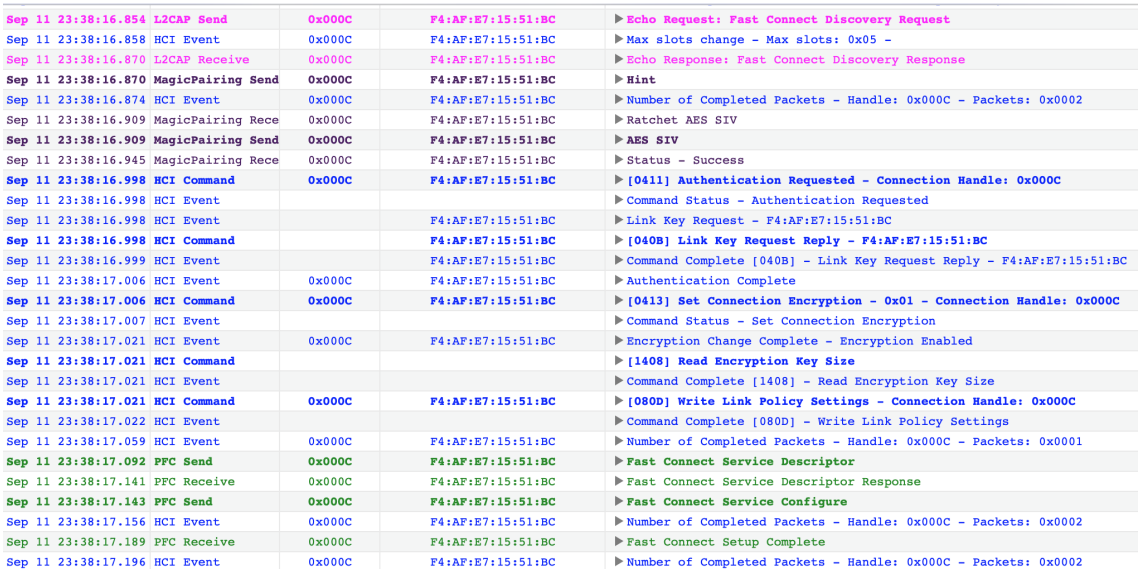
4.2.2 *Apple Bluetooth PacketLogger*

To debug Bluetooth communication on *macOS* and *iOS*, *Apple* provides a utility called *PacketLogger*. It is available for registered developers at *Apple*'s developer page[2] as part of the *Additional Tools for Xcode* package. *PacketLogger* provides a detailed log of the complete HCI communication between the host and the Bluetooth chip. This means that HCI commands and events from and to the Bluetooth chip are logged, as well as the communication between the device and other Bluetooth devices. In addition to the logging functionality, it also parses certain application protocols, including some of *Apple*'s proprietary ones. However, we found that the parsing functionality for some of the proprietary protocols has been removed in newer versions of *PacketLogger*, presumably because the functionality was never intended to be released to the public. In various instances the protocol parsing turned out to be flawed, as protocol traces from real, successful communications (e.g., between an *iPhone* and the *Apple Watch*) were not parsed correctly and result in error messages in the graphical user interface of *PacketLogger*. In some instances, the protocol is not parsed completely, i.e., only some of the fields and messages are parsed. We found that the last version that still includes the parsing capability of proprietary protocols is the one shipping with *Xcode* 10 (*Additional Tools for Xcode 10*), i.e., *PacketLogger Version 6.0.9*. In the *PacketLogger* version shipped with *Xcode* 11, these capabilities were removed. Section 4.2.2 shows a screenshot of the logging view of *PacketLogger* including two of *Apple*'s proprietary Bluetooth protocols (*MagicPairing* and *FastConnect*).

PacketLogger is an important tool for this research as it helps reverse-engineering and understanding the used Bluetooth protocols immensely, even when there is no parsing for the particular protocol implemented. As of *iOS* 13, *PacketLogger* provides a live log of the Bluetooth communication of a connected *iOS* device. The only requirement is that the device has *Apple*'s Bluetooth logging profile installed [2]. However, the live logging capability only works with the newer version of *PacketLogger* shipped with *Xcode* version 11.

In addition to the dynamic analysis capabilities that *PacketLogger* provides, the parsing code itself is a valuable resource. The *PacketDecoder* binary, included in *PacketLogger*,

2 https://developer.apple.com/download/more/



Sep 11 23:38:16.854	L2CAP Send	0x000C	F4:AF:E7:15:51:BC	▶ Echo Request: Fast Connect Discovery Request
Sep 11 23:38:16.858	HCI Event	0x000C	F4:AF:E7:15:51:BC	▶ Max slots change - Max slots: 0x05 -
Sep 11 23:38:16.870	L2CAP Receive	0x000C	F4:AF:E7:15:51:BC	▶ Echo Response: Fast Connect Discovery Response
Sep 11 23:38:16.870	MagicPairing Send	0x000C	F4:AF:E7:15:51:BC	▶ Hint
Sep 11 23:38:16.874	HCI Event	0x000C	F4:AF:E7:15:51:BC	▶ Number of Completed Packets - Handle: 0x000C - Packets: 0x0002
Sep 11 23:38:16.909	MagicPairing Rece	0x000C	F4:AF:E7:15:51:BC	▶ Ratchet AES SIV
Sep 11 23:38:16.909	MagicPairing Send	0x000C	F4:AF:E7:15:51:BC	▶ AES SIV
Sep 11 23:38:16.945	MagicPairing Rece	0x000C	F4:AF:E7:15:51:BC	▶ Status - Success
Sep 11 23:38:16.998	HCI Command	0x000C	F4:AF:E7:15:51:BC	▶ [0411] Authentication Requested - Connection Handle: 0x000C
Sep 11 23:38:16.998	HCI Event			▶ Command Status - Authentication Requested
Sep 11 23:38:16.998	HCI Event		F4:AF:E7:15:51:BC	▶ Link Key Request - F4:AF:E7:15:51:BC
Sep 11 23:38:16.998	HCI Command		F4:AF:E7:15:51:BC	▶ [040B] Link Key Request Reply - F4:AF:E7:15:51:BC
Sep 11 23:38:16.999	HCI Event		F4:AF:E7:15:51:BC	▶ Command Complete [040B] - Link Key Request Reply - F4:AF:E7:15:51:BC
Sep 11 23:38:17.006	HCI Event	0x000C	F4:AF:E7:15:51:BC	▶ Authentication Complete
Sep 11 23:38:17.006	HCI Command	0x000C	F4:AF:E7:15:51:BC	▶ [0413] Set Connection Encryption - 0x01 - Connection Handle: 0x000C
Sep 11 23:38:17.007	HCI Event			▶ Command Status - Set Connection Encryption
Sep 11 23:38:17.021	HCI Event	0x000C	F4:AF:E7:15:51:BC	▶ Encryption Change Complete - Encryption Enabled
Sep 11 23:38:17.021	HCI Command			▶ [1408] Read Encryption Key Size
Sep 11 23:38:17.021	HCI Event			▶ Command Complete [1408] - Read Encryption Key Size
Sep 11 23:38:17.021	HCI Command	0x000C	F4:AF:E7:15:51:BC	▶ [080D] Write Link Policy Settings - Connection Handle: 0x000C
Sep 11 23:38:17.022	HCI Event			▶ Command Complete [080D] - Write Link Policy Settings
Sep 11 23:38:17.059	HCI Event	0x000C	F4:AF:E7:15:51:BC	▶ Number of Completed Packets - Handle: 0x000C - Packets: 0x0001
Sep 11 23:38:17.092	PFC Send	0x000C	F4:AF:E7:15:51:BC	▶ Fast Connect Service Descriptor
Sep 11 23:38:17.141	PFC Receive	0x000C	F4:AF:E7:15:51:BC	▶ Fast Connect Service Descriptor Response
Sep 11 23:38:17.143	PFC Send	0x000C	F4:AF:E7:15:51:BC	▶ Fast Connect Service Configure
Sep 11 23:38:17.156	HCI Event	0x000C	F4:AF:E7:15:51:BC	▶ Number of Completed Packets - Handle: 0x000C - Packets: 0x0002
Sep 11 23:38:17.189	PFC Receive	0x000C	F4:AF:E7:15:51:BC	▶ Fast Connect Setup Complete
Sep 11 23:38:17.196	HCI Event	0x000C	F4:AF:E7:15:51:BC	▶ Number of Completed Packets - Handle: 0x000C - Packets: 0x0002

Figure 11: Screenshot of *PacketLogger*'s Logging View

has not been stripped of its symbols, which makes it easier to reverse-engineer. Thus, the actual parsing functions can be easily located and reverse engineered, which helps reconstructing the protocols by a considerable amount. However, the parsing code in the actual components (such as *bluetoothd*) should always be considered the more accurate source due to the inaccuracies of *PacketLogger*. Lastly, the *PacketLogger* bundle (in the aforementioned version 6.0.9) contains a file called *ACI_HCILib_2.xml*. As it is an XML file, it is human-readable and no reverse-engineering is required. This file contains documentation related to HCI commands that seem to belong to *Apple*'s Bluetooth chip *Marconi*. For example, it contains documentation for commands and protocols that are used to synchronize a pair of *AirPods* (see Chapter 3).

4.3 AIRPODS TESTING

As part of the analysis of the *RTKit* Bluetooth stack, the firmware and the hardware of the *AirPods* were analyzed. The purpose was to get a general understanding of how the *RTKit* operating system and framework works, as well as some general knowledge about the firmware and hardware of the *AirPods*. In this section, we will give a brief overview over the attack surface of the *AirPods* (Section 4.3.1), an overview over the *AirPods* firmware (Section 4.3.2), as well as the serial console that is provided by the *AirPods* case (Section 4.3.3).

4.3.1 *AirPods Bluetooth Attack Surface*

Apple managed to restrict the over-the-air attack surface of the *AirPods* drastically. A Bluetooth connection (both classic and Bluetooth Low Energy (BLE)) to the *AirPods* can only be established when the *AirPods*' Bluetooth address is known. This address is only publicly broadcasted when the button on the case is pressed, i.e., when the owner

explicitly starts pairing. When this button is pressed, the *AirPods* are discoverable on Classic Bluetooth and their Bluetooth address is embedded in their BLE advertisements. The random Bluetooth address used to send the BLE advertisement is not connectable. Even when an attacker receives a *Proximity Pairing* advertisement, they are not able to connect to the sender. This isolation is important, as the Bluetooth connection between the *AirPods* and the user's *iPhone* (or other *Apple* device) is privileged. The *AirPods* can, e.g., access and communicate with *Siri* on the connected device.

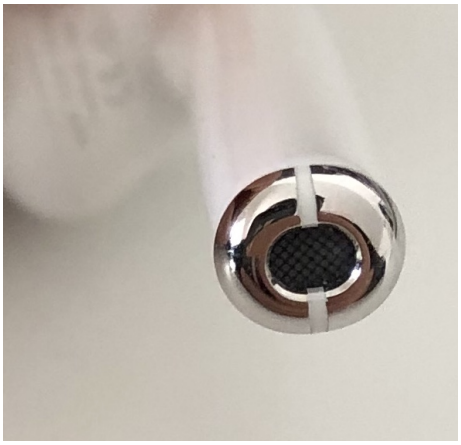


Figure 12: Interface at the Bottom of an *AirPods* Bud

In addition to the Bluetooth interface, the *AirPods* have a physical interface at the bottom of each bud. This interface can be seen in Figure 12. It is used to communicate with the *AirPods* case. The case transmits information, such as its battery state and relays the button presses using this interface. However, to use any of the case's features, an attacker would require physical access to the *AirPods*. Nonetheless, we used a logic analyzer, placed between one of the *AirPods* buds and the *AirPods* case, to intercept their communication. We found that there are regular pulses transmitted by either the *AirPods* or the case. Based on the case's serial output (see Section 4.3.3), we assume that these pulses contain the information that needs to be shared between the *AirPods* and the case. This includes battery status and version information. Apparently, *Apple* implemented a custom one-wire protocol for this purpose. This observation has also been made by Temperton [41]. Figure 13 shows a screenshot of one of the aforementioned pulses. Reverse-engineering this protocol would be beyond the scope of this work, which is why we did not investigate any further and leave this protocol for future work.

4.3.2 *AirPods Firmware*

First, the *AirPods* firmware needs to be obtained. Fortunately, the firmware update can be downloaded from *Apple*'s servers. The firmware download URLs can be found on the local filesystem of *Apple* devices that have been paired to *AirPods* before. On *macOS*, for example, there exists a folder `/System/Library/Assets`, which stores cached firmware blobs, as well as a *plist* file containing meta information about the firmwares. In case the second generation *AirPods* have been paired to a *Mac* before, the `com_apple_MobileAsset_MobileAccessoryUpdate_A2032_EA` folder exists within the `Assets` directory (`A2032` is

Figure 13: Logic Analyzer Screenshot of the *AirPods* Case Protocol

the identifier of the second generation *AirPods*). This folder might contain a cached firmware update, if the device has recently updated the firmware of connected *AirPods*. Additionally, the *plist* includes a reference and a link to the current firmware which can be downloaded. Even without owning *AirPods*, the firmwares can be easily obtained, as the *iPhone Wiki* maintains a list of firmware updates and their links [34].

The *AirPods* firmware update files differ between the first generation *AirPods* and newer generations (*AirPods Pro* and the second generation *AirPods*). The first generation's firmware file is a text file called `B188_App.txt` and contains the raw firmware data encoded in ASCII-Hex. The firmware file of the newer models is called `ftab.bin`. The *FTAB* file format is a custom firmware bundle binary format by *Apple*. In addition to the *AirPods* firmware, we found that this format is also used, for example, for the *iPhone 11*'s new *U1* chip. Note that other *RTKit* devices, such as the *Apple Pencil* or the *Smart Keyboard* use a different firmware bundle format, namely the *AFU* format [5].

In the following, the *FTAB* file format is documented in more detail. The first 32 bytes seem to be a header that is equal between different firmware versions of the AirPods. The content of the header is shown in Listing 4.1.

```
00000000  00 00 00 01 ff ff ff ff  00 00 00 00 00 00 00 00  |................|
00000010  6f 68 3d 00 be 0b 00 00  00 00 00 00 00 00 00 00  |oh=.............|
```

Listing 4.1: First 32 Bytes of an *AirPods FTAB*

The following bytes describe the content of the *FTAB* file, starting with a string identifying the type of *FTAB* (here `rkosftab` – presumably *RTKitOS FTAB*) and the number of firmware files included in the bundle, in this case `37` (`0x25` in hexadecimal).

```
00000020  72 6b 6f 73 66 74 61 62  25 00 00 00 00 00 00 00  |rkosftab%.......|
00000030  72 6b 6f 73 80 02 00 00  20 75 0e 00 00 00 00 00  |rkos.... u......|
00000040  61 6f 70 66 a0 77 0e 00  c0 f4 04 00 00 00 00 00  |aopf.w..........|
00000050  61 63 69 62 60 6c 13 00  00 00 0b 00 00 00 00 00  |acib`l..........|
00000060  70 68 79 62 60 6c 1e 00  a0 4d 00 00 00 00 00 00  |phyb`l...M......|
```

Listing 4.2: Excerpt of the Table of Contents of an *AirPods FTAB*

Afterwards, a list of files included in the *FTAB* file follows. The structure for each entry can be seen in Figure 14. It includes the filename, its offset into the *FTAB*, and its size. The table of contents in an *FTAB* file is shown in Listing 4.2.

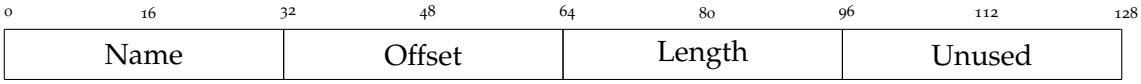


Figure 14: *FTAB* Bundle File Entry Structure

The *FTAB* file ends with an *IMG4 Manifest*. *IMG4* is a proprietary file format by *Apple* used to bundle data, such as firmware or ram-disks, along with their signatures [17]. In this case, *Apple* does not entirely rely on the *IMG4* file format, but instead bundles the *FTAB* file format with the signature part of the *IMG4* file format – the *IMG4 Manifest*. This manifest can be found by searching for the string `"IM4M"`. This also contains *Apple*'s signature root certificate *Apple Secure Boot Root CA*.

There are multiple components and protocol layers involved in updating the *AirPods* firmware. The update path begins with the *AirPods* submitting their current version information to the host device. This is done via the *AAP* protocol's `Version Information` response. The response contains information such as the device's name, model, serial number, active and pending version information, and the *External Accessory* protocol name. With this information, *bluetoothd* creates an *EASession*, which is picked up by the *accessoryd* daemon.

The firmware file is transmitted as a whole through the external accessory sub-protocol called `com.apple.accessory.updater.app.61`. To communicate using this protocol, the *AAP* protocol's `EASession Download` command is used. After the transmission of the firmware is finished, it is stored as *pending* version on the *AirPods*. Only when the *AirPods* are put back into the case and it is closed, the case will trigger the firmware update. During this, both the *AirPods* buds and the case will update themselves.

4.3.3 *AirPods Case*

The *AirPods* ear buds do not possess any connectors that allow them to be charged in their own. For this purpose, they come with a portable case that has a rechargeable battery. Additionally, it has a Lightning connector, which allows its internal battery to be charged over USB. At the bottom of an *AirPods* bud are two contacts that are connected when the bud is inserted in the case. This connection is used to charge the buds with a potential of 5 V. In addition to the charging current, *Apple* implemented a custom two-wire protocol to exchange data between the case and the buds. The battery state of the case is transmitted to the host that is connected to the *AirPods*. As the case itself does not include a separate radio, information from the case to the host is tunneled through the *AirPods*.

The case firmware file includes hints to a serial protocol available for the case. However, the included Lightning port does not usually offer a serial interface. Fortunately, there exist testing cables for *Apple* devices that are presumably used at factories to test these devices. One such cable is called *DCSD Cable* [45]. Connecting an *AirPods* case to a

computer with such a cable provides a serial console. The baud rate required for this serial connection is 230 400 Bd.

The serial console provides various commands that can be used to gather information from and to configure the *AirPods*. These commands are usually in the following format: `[COMMAND-ARG1-ARG2-...-ARGN]`, where a command is always enclosed in brackets and might have one or more arguments that are separated by hyphens. As soon as the closing bracket character is sent, the command is interpreted. Responses usually contain the command and any results enclosed in sharp brackets. In case of an error (e.g., when the command is invalid), the console responds with `<error>`. During testing, we found that some of the commands do not respond properly. An analysis of the case firmware leads us to the assumption that some commands are only available in testing or development revisions of the *AirPods* case. Other commands call a function that reads from a memory address that seems to be mapped to a peripheral bus. The function checks whether the value at this address is equal to `0xcc`. Many commands require this to be false. During testing we found that this seems to be a condition that publicly sold devices do not fulfill. However, we did not investigate this any further. In Table 6, we document a few of the available console commands. One of the commands is the `bsys-b` command. The `b` character can be substituted by either `L` or `R` (indicating the left or the right bud). Listing 4.3 shows an example usage of this command along with its response. Note that the `BA` and `CA` fields contain the primary and secondary Bluetooth addresses of the *AirPods* which have been replaced by mock addresses here.

```
1 [BSYS-L]
2 <bsys-L-BA:cafebabe1337-CA:cafebabe1338-FV:22.3.60-FU:0.0.0-HV:2.F.20-CV:1.9.0-P0-R0-
      PRI-S0:0x82-S1:0xA0-S2:0x00-L:30-T:09591-1>
```

Listing 4.3: Example Usage of the *AirPods* Case `BSYS` Command

Command	Description
`dfu`	Sets the *AirPods* case to DFU mode and triggers a firmware update/reflashing of the *AirPods*.
`dm-xxxxxxxx`	Sets the debug output flags. `[dm-11111111]` sets all flags and results in very verbose output.
`bsys-{L,R}`	System information about the respective bud.
`sr`	Print the case firmware version.
`button`	Returns the status of the button (pressed or not pressed).
`batman`	Prints battery information.
`bsv-{L,R}`	Prints the firmware version information for the respective bud.

Table 6: A Selection of *AirPods* Case Serial Commands

The *Siri Remote* 2nd Gen, which is the remote that is shipped with an *AppleTV* (generation 4 and newer) also has a *Lightning* connector on its bottom. Like with the *AirPods* case, and other *iOS* devices, the *DCSD* cable can be used to connect to the serial console of the *Siri Remote*. It requires a baud rate of 115 200 Bd and offers a similar *RTKit*-based serial

interface as the *AirPods* case. However, unlike the *AirPods* case, it offers a help command showing the possible commands, which are much fewer than on the *AirPods*. Listing 4.4 shows the output of the `help` command.

```
$ help

CLI:Command List:
help      Usage information
button    Button management
cli       CLI management
echo      Enable/Disable echo
eeprom    EEPROM management
hid       HID over UART
radio     Radio subsystem
system    Control system behavior
version   FW information
```

Listing 4.4: Output of the *Siri Remote* Serial Console Help Command

5 FUZZING BLUETOOTH PROTOCOLS

To get an overall picture of the security of *Apple*'s Bluetooth implementations, we implement *ToothPicker*, a fuzzer that is intended to be sufficiently abstract to target a wide range of the protocols presented in Section 3.5. While fuzzing can neither guarantee the absence of vulnerabilities nor make a formal assertion about the security of a protocol, it enables an overall impression of the quality of the protocol implementation. For this purpose, we evaluate two different methods, a fuzzer that sends data over-the-air and thus operates on a realistic Bluetooth connection, shown in Section 5.1, as well as an in-process fuzzer that sets up necessary data structures and calls the protocol-specific handler functions in the running *bluetoothd* process (Section 5.2). Afterwards we describe our fuzzing procedure in Section 5.3. At the end of this Chapter, in Section 5.4, we present and analyze the findings generated by *ToothPicker*.

5.1 OVER-THE-AIR FUZZING

Fuzzing over-the-air refers to a fuzzing setup where a Bluetooth connection is set up with the fuzzing target. The fuzzing payloads are then sent over this connection. While this is a simple method to start fuzzing Bluetooth protocols on the host layers, it has various drawbacks.

(-) CONNECTION TERMINATION One of the drawbacks is that the targeted host will often terminate the connection when multiple invalid messages or packets are received. Thus, a large amount of time is wasted by disconnecting and reconnecting to the targeted device. In addition to that, a terminated connection is often difficult to distinguish from a crash of the remote device.

(-) SPEED The fuzzer's speed is dependent on the physical connection between the fuzzing target and the fuzzing device. Any interferences in the connection might slow down the fuzzer.

(-) COVERAGE Another important issue is coverage. While a generation-based fuzzer creates various valid messages that might trigger different paths, there is no guarantee that it achieves a high coverage without obtaining coverage information and adapting the fuzzing payloads based on that.

However, there are also certain advantages when using an over-the-air-fuzzer.

(+) FEW FALSE POSITIVES One of the advantages of an over-the-air fuzzer is that the probability of false positives is extremely low, as the fuzzer behaves just as any other Bluetooth peripheral.

(+) PLATFORM INDEPENDENCE The fuzzer is completely independent of the targeted device's operating system or Bluetooth stack. It can be used to fuzz different implementations without much engineering effort.

Despite the drawbacks, we decided to implement a generation-based fuzzer for one of the protocols from Section 3.5, namely the *MagicPairing* protocol. The low complexity of the *MagicPairing* protocol makes it easy to implement a generation-based fuzzer that generates every possible message type. The fuzzer randomly generates valid and invalid *MagicPairing* messages based on the reverse-engineered protocol definition (see Chapter 6).

The fuzzer is implemented on top of *InternalBlue* and our testing Bluetooth stack. For protocols on fixed Logical Link Control and Adaptation Protocol (L2CAP) channels, there is no further setup required apart from connecting to the targeted device. Fixed L2CAP channels do not require an authenticated connection on *iOS*. The fuzzer then keeps sending the generated L2CAP payloads until it receives a `Disconnection Complete` Host Controller Interface (HCI) event from the targeted device. This indicates that the targeted device either disconnected due to multiple invalid received messages or due to a crash. The fuzzer then tries to reconnect to the device. The targeted device is additionally monitored using *Apple*'s *PacketLogger*. This serves as a quick way to detect if the device crashed and if the connection was terminated. Such a crash can be detected by searching for the message *Connection to the iOS device has been lost*.

Even though *PacketLogger* can be used to detect when *bluetoothd* crashes, it will not reveal any information as to why it crashed. Fortunately, *iOS* provides built-in crash reporting. When any component on the system crashes, *iOS* creates a detailed crash report and stores it on the device. These reports can then be used to investigate any occurred crash. During operation, we discovered that the `Disconnection Complete` event is fairly unreliable. In multiple occasions the connection was terminated, but the fuzzer did not receive the event, even though the disconnection command can be seen in the *PacketLogger* trace. This drastically undermines the efficiency of the fuzzer as it needs to estimate when a connection is terminated in case it did not receive the disconnection event.

Despite the mentioned issues, the fuzzer ended up finding multiple bugs in the protocol implementations. It was able to identify these vulnerabilities as the protocol is very simple and the implementations are flawed. However, it would have to run for a very long time to uncover more complex paths and vulnerabilities within a protocol. A description of the identified vulnerabilities can be found in Chapter 6.

5.2 IN-PROCESS FUZZING

The over-the-air fuzzing approach shows two major drawbacks. Firstly, it depends on a physical connection that is not reliable under the load of the fuzzing input and requires frequent reconnects. Secondly, there is no feedback about coverage that is achieved with the generated inputs.

To improve the reliability and efficiency of the fuzzing efforts, an in-process, coverage-guided fuzzer is implemented to fuzz the reception handlers of interesting Asynchronous Connection-Less (ACL) and L2CAP-based protocols. While the throughput of messages

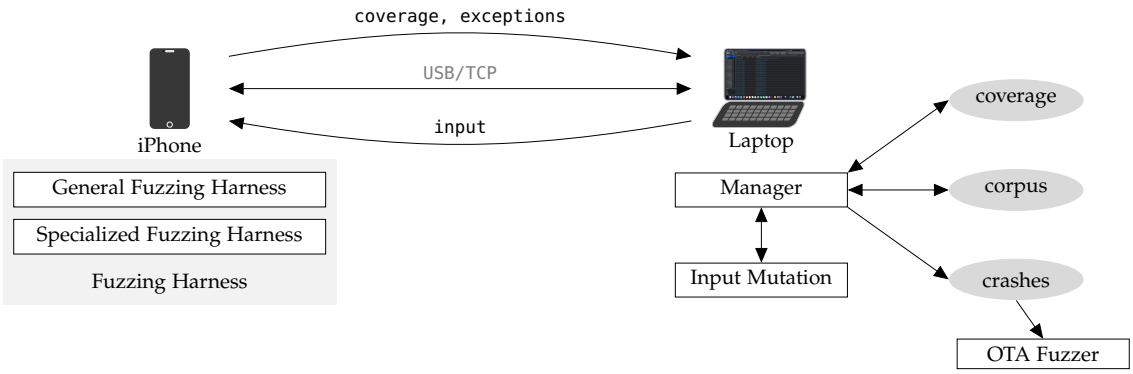


Figure 15: Architecture of the In-Process Fuzzer

and the stability of the payload delivery is much higher than in the over-the-air implementation, the in-process fuzzer comes with a different set of drawbacks.

(-) MORE FALSE POSITIVES The probability of false positives increases in comparison to the over-the-air fuzzing approach. The usual operation of the Bluetooth daemon is altered, which can lead to unexpected behavior or crashes that are not related to the fuzzing payload, but rather due to the injected fuzzing harness.

(-) PLATFORM DEPENDENCE Unlike the over-the-air approach, the in-process fuzzer is not inherently platform independent. Injecting and preparing the fuzzer inside the targeted process differs significantly for different operating systems. Details, such as function addresses, even change between versions of the same operating system.

We try to overcome the increased amount of false positives by minimizing the amount of crashes related to the injected fuzzing harness code. This is done by practically observing any side-effects during fuzzing and patching the affected functions. Nonetheless, it is not possible to guarantee a 100 % correct behavior of *bluetoothd* with the fuzzing harness embedded. In the scope of this thesis, solving the lacking platform independence is not strictly necessary. With our work we present an implementation of the in-process fuzzing approach for *iOS*. Implementations for other operating systems follow the same logic and may be implemented in future work.

5.2.1 *General Architecture*

In the following, more technical details about the fuzzer are given. We begin by describing the general setup and architecture. Then, we show some of the patches that were required to improve its reliability. Figure 15 displays the general architecture of the fuzzing setup.

In general, the fuzzer consists of two components, the *manager*, running on a computer, and the fuzzing *harness* running on the targeted *iOS* device. The responsibilities and tasks of these components are as follows:

MANAGER The manager is responsible for starting and maintaining the fuzzing process. It injects the fuzzing harness into the targeted process and handles the communication with it. In addition to that, it maintains a set of crashes that occurred during

fuzzing, a corpus to derive inputs from, and the coverage that was achieved during fuzzing. New inputs are created by the input mutation component. Based on a seed it randomly mutates data from the corpus to derive new input data.

HARNESS The fuzzing harness consists of two parts: a general fuzzing harness and a specialized fuzzing harness. The general fuzzing harness is responsible for all general operations required for fuzzing *bluetoothd*. It can create virtual connections and applies the necessary patches that are required for a stable fuzzing process. It also provides the means to collect code coverage and receives the fuzzing input from the manager. The specialized fuzzing harness, on the other hand, is specific for the targeted function and protocol that is to be fuzzed. It is responsible for preparing the received input and calling the function handler, as well as any other preparation that is needed in order to be able to fuzz the specific protocol reception handler function.

The fuzzer is initialized by providing an initial corpus of valid protocol messages. More specifically, this corpus consists of function arguments of the protocol handlers that are to be fuzzed. The fuzzer then starts to collect the initial coverage by sending the initial corpus to the fuzzing harness, which, by using the specialized harness, executes the payloads. The collected coverage is then returned to the manager, which stores it for later use.

Once the initial coverage is collected, the actual fuzzing can begin. One iteration works as follows: The manager picks one of the entries in the corpus and sends it, together with a seed value to the input mutator. The mutator then mutates the input and sends it back to the manager. The manager then proceeds by sending the input to the specialized fuzzing harness. If desired, it can mutate the input further. This is useful in cases with fields in the input that need to have deterministic values or length fields that, for certain cases, should be correct. The specialized fuzzing harness can modify the input and send it back to the manager. Sending back the modified input before actually calling the function under test is required for cases where the target crashes as a result of the input. As the injected harness crashes together with the target, the modified input would be lost. Once the manager receives the modified input, the target function can be called with this input. Usually, the protocol reception handlers within *bluetoothd* run in a separate reception thread (*RXLoop* on *iOS*). The fuzzing harness runs in its own thread. This has the advantage that a custom exception handler can be implemented for this thread. This handler can catch the exception (i.e., what would have otherwise been a crash) and terminate gracefully without crashing *bluetoothd*. While the function is called, the harness is collecting basic block coverage. After calling the function, there are three possible outcomes:

ORDINARY RETURN The function was executed successfully and returns. The collected coverage information is sent to the manager.

EXCEPTION The function was not executed successfully and results in an exception. This exception is caught and returned to the manager. The manager stores both the input and the exception type as a crash.

UNCONTROLLED CRASH In case the target (i.e., *bluetoothd*) crashes in a thread not controlled by the fuzzing harness, it will crash and generate a crash report. In this case, the exception cannot be sent to the manager. However, the manager detects a crash and can store the generated input as a crash. The corresponding crash report can be manually gathered from the operating system, in case it was generated successfully.

Even in the case of an *exception*, the input might have still lead to a false positive. Therefore, it is important to verify the identified crashes. This is done by using the over-the-air fuzzer presented in Section 5.1. As all the inputs that lead to a crash are stored by the manager, they can be forwarded to the over-the-air fuzzer. The over-the-air fuzzer can then create a real connection to the fuzzing target and send the input. As this would disrupt the flow of the in-process fuzzer, the validation must be done while the in-process fuzzer is not running. As with the general operation of the over-the-air fuzzer, the payload should be sent while monitoring the device with *PacketLogger*. In case of a crash, *iOS*'s crash logs can be examined to determine the cause. Section 7.1 shows an overview of crashes and bugs that were found using this technique.

5.2.2 *Implementation*

We implement our fuzzer based on *frizzer* [8]. It provides the basic fuzzing architecture, like coverage collection, corpus handling, input mutation, and thus a large part of the manager component. Our fuzzer, like *frizzer*, is built on FЯIDA, which is a "[d]ynamic instrumentation toolkit" [7]. With FЯIDA, custom code can be injected into a target process in the form of JavaScript code. Thus, the fuzzing harness is implemented in JavaScript and injected into *bluetoothd*. The master is implemented in Python by using FЯIDA's Python bindings. The test case generator *radamsa* [15] acts as the input generation component.

5.2.2.1 *L2CAP Protocol Handlers*

Usually, the function definitions of fixed L2CAP channel reception handlers in *bluetothd* look as shown in Listing 5.1. The first argument is a pointer to a connection structure, the second argument is a pointer to the received data, and the last argument is the length of the received data.

```
1  void l2cap_reception_handler(void *connection, char *data, size_t len)
```

Listing 5.1: Function Signature of L2CAP Reception Handlers in *bluetoothd*

In the first version of the fuzzer, the strategy to fuzz a specific L2CAP reception handler was as follows:

- Hook the targeted reception handler function,
- create a physical Bluetooth connection (e.g., using a device with *InternalBlue*),
- send real L2CAP data to the targeted protocol,

- copy the pointer to the connection structure, and
- call the reception handler with the copied connection structure.

Once the pointer to the connection structure is stored, the fuzzing harness can start calling the reception handler with the stored connection and arbitrary L2CAP data. While this is a simple approach to ensure that the proper data structures for a connection are in place, it has similar drawbacks as the over-the-air variant of the fuzzer. Firstly, a real physical connection is required, and secondly, when one of the peers decides to terminate the connection, the allocated connection structure is destroyed and a new connection has to be created. This also implies hooking the function again and storing a new connection structure. Instead, we decided to virtualize the connection required for fuzzing. This can be done by calling the function that allocates the connection structure from our fuzzing harness.

5.2.2.2 *Virtual Connection Creation*

An example of one of the tasks the specialized harness has to handle is creating a forged ACL connection. While reverse-engineering *bluetoothd* on iOS, we found a function that is used to allocate exactly this connection. Due to the lack of symbols we called this function `allocateACLConnection`. This function can now be called using the specialized harness to create a forged ACL connection and corresponding handle. As function symbols are missing, the relative address of the function needs to be extracted from each *bluetoothd* version that is under test. During this thesis we were mainly fuzzing *bluetoothd* on *iOS* 13.3 on an *iPhone* 7. The function addresses used in the following code snippets are all taken from this configuration. As an example, the relative address of the `allocateACLConnection` function for this configuration is `0xc81a0`. The function accepts two arguments, the Bluetooth address of the peer and another value that is stored directly in a field of the ACL connection structure. While reverse-engineering and dynamically analyzing this structure, we found that this seems to be a field indicating the status of the handle. This status is mostly set to the value `0` when an L2CAP reception handler is called. An example of how such a forged ACL connection can be created in FЯIDA is shown in Listing 5.2.

```
// create a buffer for the Bluetooth address
var bd_addr = Memory.alloc(6);
// resolve function address
var base = Module.getBaseAddress("bluetoothd");
// 0xc81a0 is the offset to the allocateACLConnection function
var fn_addr = base.add(0xc81a0);

// create reference to function to be able to call it from Javascript
var allocateACLConnection = new NativeFunction(fn_addr, "pointer", ["pointer","char"]);
// write the Bluetooth address to memory
bd_addr.writeByteArray([0xca, 0xfe, 0xba, 0xbe, 0x13, 0x37]);

// call the function and create a forged ACL connection
// if handle is != 0 then the call was successful
var handle = allocateACLConnection(bd_addr, 0);
```

Listing 5.2: Creating a Forged ACL Handle Using FЯIDA

For most of the L2CAP-related reception handlers, this handle is the first argument. In the case of the *MagicPairing* reception handler, the arguments are the handle, a pointer to the received data, and the length of the received data. Using this knowledge, a FЯIDA script that calls the *MagicPairing* reception handler can be implemented. An example of this, continuing the script from Listing 5.2, is shown in Listing 5.3. Note that the relative address of the *MagicPairing* reception handler is `0x129110` on *iOS* 13.3 on an *iPhone* 7.

```
// allocate a 128 byte buffer to hold the MagicPairing data
var payload_buffer = Memory.alloc(128);

// like before, create a reference to the function to call it from JavaScript
// 0x129110 is the offset to the MagicPairing reception handler function
var mp_fn_addr = base.add(0x129110);
var magic_pairing_handler = new NativeFunction(mp_fn_handler,
    "void", ["pointer", "pointer", "int64"]);

// create an array to form our input data
var input_array = [0xff, 0x00, 0x00, 0x00];

// before calling the handler the data has to be written into the buffer
Memory.writeByteArray(payload_buffer, payload_array);

// finally call the MagicPairing handler with the handle and the data we created
magic_pairing_handler(handle, payload_buffer, parseInt(payload_array.length));
```

Listing 5.3: Calling the *MagicPairing* Reception Handler Using FЯIDA

Based on these scripts, many of the fixed channel L2CAPprotocols in *bluetoothd* can be fuzzed. Similar to creating an ACL handle, a Bluetooth Low Energy (BLE) handle can be created. This involves calling the `allocateLEConnection` function. Again, this name was chosen arbitrarily due to the lack of symbols.

5.2.2.3 *Stabilizing Virtual Connections*

Even though the connection that is created is virtual, it can still be disconnected. During our analysis of *bluetoothd* we identified two functions that seem to have the purpose to disconnect or destroy an ACL/BLE connection. We can overwrite these functions, to prevent our forged connections from being disconnected. One of these functions is called `OI_HCI_ReleaseConnection`[1]. Listing 5.4 shows how this function can be overwritten using FЯIDA.

While hooking and replacing these functions prevents the connection structures from being destroyed, this disconnection prevention technique cannot be used for the over-the-air fuzzer. During an over-the-air fuzzing session, there exist four distinct representations of the connection, two for each of the peers. The first connection representation are the data structures that the Bluetooth stack (like the *ACL connection* in *bluetoothd*) creates. These usually store information about the controller's HCI handle and other application-layer information, such as open L2CAP channels. The other representation of the connection resides within the Bluetooth chip. The chip allocates an HCI handle the stack can use

1 Strings inside the binary hint that this is the actual name of the function.

```
var release_fn_addr = base.add(0x0c87e0);

// replace the OI_HCI_ReleaseConnection function with our own function
Interceptor.replace(release_fn_addr,
    new NativeCallback(function(a, b) {
        // do nothing and just return
        return;
    }, "void", ["pointer"])
);
```

Listing 5.4: Overwriting `OI_HCI_ReleaseConnection` Using FЯIDA

to reference the connection. The chip also holds additional state to keep the connection alive. Even if we manage to prevent *bluetoothd* from destroying the connection, we cannot easily control the other involved components. Thus, the in-process fuzzing variant with the virtual connection is the one that can be controlled best.

5.2.2.4 *Creating Virtual BLE Connections*

Some of the chosen protocols require a BLE connection. While the generic ACL reception handler is the same for Classic Bluetooth and BLE, the virtual connection creation differs. Firstly, there is a different function to create a BLE connection. In this thesis we refer to this function as `allocateLEConnection`. Owed to the fact that the BLE connection setup is more complex and has more parameters to configure, the `allocateLEConnection` function accepts 8 parameters. The other difference is that it does not have a return value and thus does not return a pointer to the connection structure. To obtain this structure, another function has to be called. We call this function `getConnectionStruct`. Its only parameter is the handle value of the respective Bluetooth connection (i.e., the one that is used to reference a connection when communicating with the Bluetooth chip). As this value can be chosen arbitrarily when calling the `allocateLEConnection` function (as long as it does not exist yet), we can use the chosen value as parameter to the `getConnectionStruct` function to obtain the connection structure. The last step that needs to be done is creating a *ReadRemoteVersionInformation* HCI event. This concludes the BLE connection setup. Fortunately, there exists the function `ReadRemoteVersionInformationCB`, which is a callback function that gets called when such an HCI event is received from the Bluetooth chip. We can directly call this function to simulate the reception of an *ReadRemoteVersionInformation* HCI event. To obtain the various arguments required for the `allocateLEConnection` and `ReadRemoteVersionInformationCB` functions, we hooked the respective functions, created a real BLE connection and stored the parameters. Listing 5.5 shows an excerpt of the fuzzing harness' forged connection setup for BLE.

5.2.2.5 *Opening Dynamic L2CAP Channels*

Different from fixed L2CAP channels, dynamic L2CAP channels need to be opened and allocated before they can be used. This involves sending multiple requests over the *Signal Channel*. First, a *Connection Request* command is sent to the remote device. This requests includes the local device's Channel ID (CID) as well as the Protocol/Service Multiplexer (PSM) specifying the protocol that is carried over the L2CAP channel. On success, the remote device responds with a *Connection Response*. In the case that the remote device

```
var allocateLEConnection_fn_ptr = base.add(0x11ba24);
var ReadRemoteVersionInformationCB_fn_ptr = base.add(0x12117c),
var getConnectionStruct_fn_addr = base.add(0x0c7df4),

// observed by hooking the allocateLEConnection function
var x0 = 0x0, x2 = 0x1, x4 = 0x24, x5 = 0x0, x6 = 0x1f4, x7 = 0x1;
// the handle value for the BT connection (e.g., 0x40)
var handle_value = 0x40;
var bd_addr_buffer = Memory.alloc(8);

var allocateLEConnection = new NativeFunction(allocateLEConnection_fn_ptr,
    "void", ["long", "long", "int", "pointer", "long", "int16", "int16", "int8"]);
var getConnectionStruct = new NativeFunction(getConnectionStruct_fn_ptr,
    "pointer", ["int"]);
var ReadRemoteVersionInformationCB = new NativeFunction(
    ReadRemoteVersionInformationCB_fn_ptr,
    "void", ["long", "pointer", "long", "long", "long", "pointer", "long", "long"]);

// the buffer consists of the BT address type (0x01 -> random), the Bluetooth address
// and the state (which was a separate argument in the Classic connection creation
// function)
bd_addr_buffer.writeByteArray([0x01, 0xca, 0xfe, 0xba, 0xbe, 0x13, 0x37, 0x00]);

allocateLEConnection(x0, handle_value, x2, bd_addr_buffer, x4, x5, x6, x7);

// now that we have the handle we need to fetch the connection struct
var connection = getConnectionStruct(handle_val);
if (connection == ptr(0x00)) {
    console.error("No connection for this handle, something in BLE setup went wrong");
}

// fake a ReadRemoteVersionInformation event by calling its callback function
// again, the values are taken from a real BLE connection
x0 = 0x0, x2 = 0x8, x3 = 0x46, x4 = 0x1130, x5 = ptr(0x0), x6 = 0x0, x7 = 0x403
ReadRemoteVersionInformationCB(x0, connection, x2, x3, x4, x5, x6, x7);
```

Listing 5.5: Creating a Forged BLE Connection Using FЯIDA

does not know a protocol corresponding to the sent PSM value, a *Command Reject* is sent by the device. To create a dynamic L2CAP channel for the Service Discovery Protocol (SDP) protocol, a *Connection Request* to PSM `0x0001` needs to be sent.

We identify a function that handles the dynamic L2CAP channel creation to prevent the overhead of sending, receiving, and parsing multiple *Signal Channel* frames. The function `OI_LP_ConnectionAdded`[2] can be called to allocate a dynamic L2CAP channel for a given PSM. An example of how to call this function is shown in Listing 5.6. Note that this function does not return the allocated CID. Thus, we need to hook the function itself and extract the CID value after it has been allocated. Following that, *bluetoothd* will be able to receive L2CAP data for the protocol the channel has been opened for. To fuzz the protocol, the general ACL or L2CAP reception handlers can be called by specifying the CID.

2 Debug strings indicate that this is the name of the function.

In Appendix A.2, we provide an overview of all the functions we identified that are useful for fuzzing *bluetoothd*. We provide their offset addresses on *iOS* 13, as well as their function signatures.

```
// 0xf02b8 is the offset of the OI_LP_ConnectionAdded function
var fn_addr = base.add(0xf02b8);
var _OI_LP_ConnectionAdded = new NativeFunction(fn_addr, "void", ["pointer", "int",
    "int", "int", "int"]);

// In the OI_LP_ConnectionAdded function there is a print statement which prints
// the PSM and the CID right after it's allocated. At this stage, the L2CAP channel
// structure is in register x0, so we can extract the CID from there.
// 0xf0508 is the offset of the print statement in the OI_LP_ConnectionAdded function
var cid = null;
var listener = Interceptor.attach(base.add(0xf0508), function() {
    if (cid == null) {
        cid = Memory.readShort(this.context.x0.add(4))
    }
});

var x0 = handle;        // connection handle pointer
var x1 = PSM;           // the PSM of the protocol we want to connect to
var x2 = 0x40;
var x3 = 0x1;
var x4 = handle_value;  // the connection handle value
_OI_LP_ConnectionAdded(x0, x1, x2, x3, x4);

// wait until the CID is set by the interceptor function
while(cid == null) {
    sleep(100);
}
// detach the interceptor again
listener.detach();
```

Listing 5.6: Allocating a Dynamic L2CAP Channel Using FЯIDA

5.3 FUZZING PROCEDURE

Now that we have established the architecture and overall principle of the fuzzer, the particular set up and the procedure used during this thesis are presented. An overview of the targeted protocols, chosen based on the results in Section 3.5, is shown in Table 7. In the following, the reasoning for the chosen protocols, the generation of the corpora, as well as the optimizations are explained. In total, we chose *eight* different protocols to fuzz. The first protocol, *MagicPairing*, additionally serves as a validation for the in-process fuzzing methodology. If it does find the same crashes we were able to find with the over-the-air method, we can assume that it is at least as effective.

The second column describes how the corpus for the particular protocol was generated. The corpora are mostly generated by intercepting (i.e., *recording*) both the ACL reception and transmission functions while the specific data is sent over a real physical connection. In cases where this is not possible the corpus is generated manually. For the *LEAP*

Protocol	Corpus	Optimization	Technology
MagicPairing	- record of *AirPods* pairing - manually created *Ping* message	- correct ACL length - correct L2CAP length - no fragmentation	Classic
ACL	- record of L2CAP Echo Requests and Responses	none	Classic
BLE ACL	- manually created BLE Signal Channel messages	none	BLE
GATT	- record of interaction with GATT exploration App	- correct ACL length - correct L2CAP length - no fragmentation	BLE
Magnet	- connect and disconnect *Apple Watch*	- correct ACL length - correct L2CAP length - correct *Magnet* length - keep track of protocol version - no fragmentation	BLE
LEAP	- manually created by reverse engineering	- correct ACL length - correct L2CAP length - no fragmentation	BLE
L2CAP Signal Channel	- record traffic by connecting and disconnecting *AirPods* - record *FastConnect* Discovery messages	- correct ACL length	Classic
SDP	- record traffic by connecting to *macOS* and query device with *Bluetooth Explorer*	- correct ACL length - correct L2CAP length - no fragmentation - correct SDP length	Classic

Table 7: Fuzzing Plan

protocol, we did not have a device that speaks this protocol. Thus, the corpus was created by reverse-engineering the protocol and manually generating valid messages. In case of the *MagicPairing* protocol, we augment the recorded corpus with a manually created message type, that would otherwise not be included as it does not occur regularly in a real connection.

The third column describes the optimization that the specialized harness applies to the mutated fuzzing input. In most cases, there are three optimizations. The correction of the L2CAP length field, the optimization of the ACL length field, and the flags in the handle that determine the L2CAP fragmentation. While correcting the length field removes the fuzzing input's randomness, it ensures that the generated input packets do not unnecessarily fail at length checks and rather reach deeper into the actual parsing code. In case of the *Magnet* protocol, we implement two additional optimizations. Firstly, *Magnet*'s own length field is correctly calculated, and secondly, the fuzzer keeps track of which version is currently negotiated. This is important for correcting the length field, as the offset and the size of the *Magnet* length field changes. Keeping track of the current version is as simple as monitoring the generated packets. As soon as a *Version* packet is sent, the specialized harness adapts the following version fields accordingly.

The last column indicates the Bluetooth technology (i.e., Classic Bluetooth or BLE). Depending on the technology the harness has to adapt the creation of the virtual connection.

We wrote an additional script that intercepts various logging functions and prints the messages to monitor the fuzzing process. While *iOS*'s standard logging (e.g., via the `Console.app` on *macOS*) could be used to monitor the logs, our approach has the advantage that even internal debug logging, that would not be visible, can be made visible. Before some of *bluetoothd*'s logging calls, there are checks that determine whether it is an internal build. We patched *bluetoothd* in such a way that it assumes it is an internal build to increase the verbosity even more. Monitoring the logs turns out to be useful to observe certain behavior. During the initial fuzzing rounds of the *Magnet* protocol, we observed that the majority of logging messages were concerning the invalid length field in the packet. This lead us to the two additional optimizations for the *Magnet* protocol.

Another issue that arises due to the in-process fuzzing is that the resource utilization of *bluetoothd* is much higher than usual. *Jetsam*, *Apple*'s out-of-memory killer can be configured to kill processes when they take up more ressources than they should have [27]. To counter this problem, we adapted the *Jetsam* configuration file[3] file to increase both *bluetoothd*'s memory limit and priority. This leads to much fewer cases where *bluetoothd* is terminated due to resource consumption.

5.4 FUZZING RESULTS

An overview of identified vulnerabilities is shown in Table 8. Note that this table does not cover vulnerabilities within the *MagicPairing* protocol. These are presented in the dedicated Chapter 6. In the following, more details about the vulnerabilities are provided. The payloads as well as additional information on triggering the crashes and bugs can be found in Appendix A.1.

ID	Description	Effect	Detection Method	OS	Disclosure	Status
L2CAP1	L2CAP Zero-Length	Crash	Over-the-Air	*RTKit*	Dec 4 2019	Not fixed
L2CAP2	L2CAP Groups	Crash	In-Process	*iOS*	Mar 13 2020	Not fixed
SMP1	SMP OOB	Partial PC Control	In-Process	*iOS*	Mar 31 2020	Not fixed
SIG1	Missing Checks	Crash	In-Process	*iOS*	Mar 31 2020	Not fixed

Table 8: List of Identified Vulnerabilities

L2CAP1: L2CAP Zero-Length

While fuzzing *MagicPairing* over-the-air, we identified a crash in the *RTKit* Bluetooth stack, more specifically, the *AirPods 1* and *2*. When sending an L2CAP message with the length field set to zero and no payload, the *AirPods* crash. As there are no publicly

3 `/System/Library/LaunchDaemons/com.apple.jetsamproperties.D20.plist`, where `D10` is the identifier of the *iPhone 7* we had as a test device. For other devices, this identifier needs to be adapted.

documented debugging capabilities for the *AirPods*, it is not possible to tell whether the Bluetooth thread or the whole operating system crashes. We observe that the music stops playing, the connected *iPhone* reports the *AirPods* as disconnected, and after a few seconds, the *AirPods* play a sound indicating a successful connection.

L2CAP2: L2CAP Groups

This crash is another `NULL` pointer dereference, albeit more severe than the previous ones. It is accessible via both BLE and Classic Bluetooth and is part of *L2CAP Group* feature. This is indicated by logging messages in the crashing function that mention the file `corestack/l2cap/group.c`. However, the *L2CAP Group* feature is no longer supported since the Bluetooth 1.1 specification. We assume the group reception function has been accidentally left in the code. Depending on the data that is received, it tries to find a matching entry in a function table allocated on the heap. While the table is allocated, it never gets initialized. Thus, all its entries are zero. When the payload starts with a `NULL` byte, the first entry is identified as matching entry. The code then tries to jump to the function pointer stored in that table entry, which also is a `NULL` pointer. Any control over this table would immediately result in control over the instruction pointer. In addition to an *iPhone 7* on *iOS 13.3*, we were able to reproduce the crash on an *iPad 2* with *iOS 9.3.5* (released on August 25 2016), as well as an *iPhone 4* on *iOS* 9.3.5 (released on November 10 2011). While the crash is not critical per se, it shows how long the *iOS* Bluetooth stack has not been tested thoroughly. As *iOS 9* and *iOS 5* still had another Bluetooth stack architecture, the crash is within `BTServer` instead of `bluetoothd`.

SMP1: Security Manager Protocol Out-of-Bounds Jump

This vulnerability occurs in the reception handler of the Security Manager Protocol (SMP), which, as of *iOS* 13, is accessible via both Classic Bluetooth and BLE. The cause for this flaw is an incorrect check of the received protocol opcode value, which leads to an out-of-bounds read. The value that is read is treated as a function pointer. Listing 5.7 shows a pseudocode representation of the flawed opcode check. If the opcode `0x0f` is sent, the bounds check is still valid. However, the global function table where the handlers for the specific opcodes reside only has `15` entries. As the table is indexed by zero, accessing the 15th entry will lead to a read out of bounds. The table is immediately followed by other data. As the tables for Classic Bluetooth and BLE are at different locations, the result is different, depending on which technology is chosen. We found that on *iOS 13.3*, the data following the Classic Bluetooth SMP function table is partially controllable by an attacker. More speifically, it is followed by two *four-byte* values, an unknown value that we observed to be `0x00000001` and a global counter that is incremented for each Bluetooth connection that is created. Due to its layout, the connection counter determines the Most Significant Bits (MSBs) of the address. Therefore, an attacker could crash *bluetoothd* using one of the crashes we identified and then create a specific amount of connections to form the address. However, we did not find a way to influence the other value. This leads to the lower four bytes of the address to be equal to `0x00000001`, which is an unaligned address and thus crashes *bluetoothd*. In case it is possible to influence this value, or the layout of the memory following the function table changes, this flaw could potentially lead to a

control flow hijack. However, the likelihood of this flaw being usable in a Remote Code Execution (RCE) attack is very low. The amount of preparation required for abusing this flaw is very high. This includes preparing or determining a useful address to jump to and also being able to precisely control the value.

```
# get opcode from the input data
opcode = data[0]
# flawed opcode check here
if (opcode <= 0xf):
    # the handler is resolved from the function table
    handler = GLOBAL_FUNCTION_TABLE[opcode]
    # the code checks if the resolved pointer is not null
    if handler != None:
        handler(...)
```

Listing 5.7: Pseudocode of the Flawed SMP Opcode Check

As shown in Table 7, we did not explicitly target the SMP protocol. The *SMP1* vulnerability was found by the ACL fuzzing pass which had a corpus of only L2CAP Signal Channel frames. This also shows that the input generation and mutation mechanism is capable of identifying new paths and protocol messages.

SIG1: Signal Channel Missing Checks

The Signal Channel fuzzing pass identified three crashes that all originate from the same flaw. These crashes are a result of accessing memory at the location `0x00`, `0x08`, and `0x10`. The crash occurs within the *Configuration Request*, and *Disconnection Response* frame handlers in the Signal Channel. These frames are all related to dynamic L2CAP channels. Thus, they contain a CID. To obtain the necessary data structure for an L2CAP channel with a given CID, a lookup is done via the `ChanMan_GetChannel` function. This returns the channel data structure for both dynamic and static L2CAP channels. If the L2CAP channel is a dynamic channel, the data structure contains a field which points to another structure with additional information about the channel. In case of a fixed L2CAP channel this additional data structure does not seem to exist and the field value is `0`. The Signal Channel parsing code in *bluetoothd* does not differentiate between static and dynamic L2CAP channels. Thus, the code falsely treats this null pointer as pointer to a structure and dereferences the fields at offset `0x00` `0x08`, and `0x10`, leading to crashes due to invalid memory accesses. A pseudocode representation of this flaw is shown in Listing 5.8. The example is taken from the reception handler of the *Disconnection Response* frame handler.

```
// the L2CAP channel structure is obtained from the received CID
status = ChanMan_GetChannel(channel_id, &channel);
// if this CID exists, the dynamic structure is obtained from the channel structure
if (status == 0) {
    dynamic_structure = get_dynamic_structure_from_channel(channel);
    // the dynamic_structure pointer is not checked before dereferencing it, thus
    // this line will try to dereference the value 0x10 which leads to a crash
    if (*(char *)(dynamic_structure + 0x10) == 0x06 && [...]) {
        [...]
    }
}
```

Listing 5.8: Pseudocode of the Signal Channel Structure Pointer Dereference

6 MAGICPAIRING

MagicPairing is a proprietary protocol by *Apple* that is used to provide seamless pairing capabilities between a user's *AirPod*s and all their *Apple* Devices. This is enabled by synchronizing keys over *Apple*'s cloud service *iCloud*. The ultimate goal of the *MagicPairing* protocol is to derive a Bluetooth link key that is used by the master device and the *AirPod*s. This means that the protocol, which is implemented on top of Logical Link Control and Adaptation Protocol (L2CAP) is used before an encrypted Bluetooth connection is established. Another benefit of using *MagicPairing* to derive a Bluetooth link key is that a fresh Key is used for each connection, which greatly reduces the lifetime of this link key.

When a new or reset pair of *AirPods* is initially paired with an *Apple* device belonging to an *iCloud* account, Secure Simple Pairing (SSP) is used. All subsequent connections between the *AirPods* and devices connected to that *iCloud* account will use the *MagicPairing* protocol as pairing mechanism. *MagicPairing* involves multiple keys and derivation functions. It relies on AES in SIV mode [13] for authenticated encryption.

6.1 PROTOCOL DESCRIPTION

The protocol is comprised of mainly five phases that are described more thoroughly in the following sections. As the protocol is not publicly documented, the following naming relies on logging output and strings found in the respective components, i.e., the Bluetooth daemon *bluetoothd* for *iOS*, *macOS*, and the *AirPods* firmware.

6.1.1 *Protocol Phases*

The *MagicPairing* protocol depends on a shared secret between the two participants. Therefore, the first phase establishes and exchanges a secret. The following phases then determine steps in the protocol itself. In the following explanations, we assume a protocol flow with the *AirPods*. However, the protocol does not explicitly need to target *AirPods* and can potentially be used with any other Bluetooth device. The protocol flow is visualized in Figure 16.

PHASE 1: KEY CREATION AND DISTRIBUTION

MagicPairing relies on a shared secret between the *AirPods* and a user's *iCloud* devices, the *Accessory Key*. This key is created by the device that is the first to pair with the *AirPods* for a specific *iCloud* account. After establishing an encrypted Bluetooth connection using SSP, the *Accessory Key* needs to be transmitted to the *AirPods*. This is done by using the *AAP* Protocol. In addition to the *Accessory Key*, the host also creates an *Accessory Hint* which uniquely identifies the connection between an *iCloud* account (along with its connected devices) and the *AirPods*. To create the *Accessory Key* and the *Accessory Hint*,

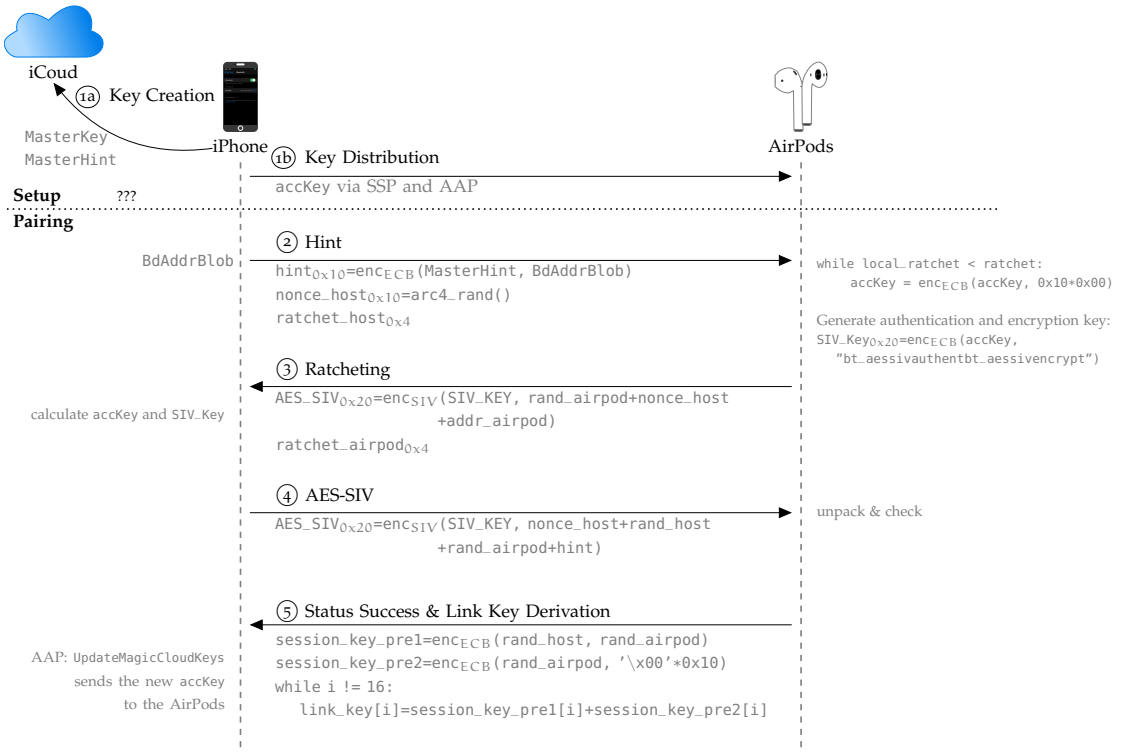


Figure 16: *MagicPairing* Protocol Steps

the initiating device uses the *iCloud* account's *Master Key* and *Master Hint*, respectively. In case these *Master* keys do not exist yet, the device provisions them by creating random bytes. Another component that is needed to create the *Accessory Key* and *Accessory Hint* is the so-called *Bluetooth Address Blob*, which is a deterministic mutation of the Bluetooth address of the targeted device. The mutation is shown in Listing 6.1.

```
1  blob[1:5] = address[5:0]
2  blob[6:9] = address[1:4] ^ address[0:3]
```

Listing 6.1: Creating a *MagicPairing* Bluetooth Address Blob

To create the *Accessory Key*, the *Bluetooth Address Blob* is encrypted with the *Master Key* using AES in ECB mode. The *Accessory Hint* is created by encrypting the *Bluetooth Address Blob* with the *Master Hint*, respectively.

PHASE 2: HINT PHASE

The first phase in the actual *MagicPairing* protocol is the *Hint* phase. Usually, the host initiates the pairing by sending a *Hint* message to the *AirPods*. The *Hint* message includes three entries, the *Hint*, a random nonce generated by the initiating host and a *Ratchet*, which is a counter that is used in later steps of the pairing process for rotating keys. The receiving end will then use a local table to look up if it has an *Accessory Key* for the connecting device. The *AirPod*s use the *Hint* that is included in the *Hint* message as a reference, *iOS* and *macOS* devices use the connecting device's Bluetooth address to look up the key. If a key is found, the *Ratcheting Phase* begins. If not, a *Status* message is sent

back indicating that the initiating device is unknown to the targeted device (*No Keys for Peer*, `0x08`).

PHASE 3: RATCHETING PHASE

The *Ratcheting Phase* is essentially a key rotation and derivation phase. First, the *Accessory Key* is rotated and then a *SIV Key* is derived from the rotated key. The *Accessory Key* is rotated by encrypting a buffer of 16 null-bytes with the current *Accessory Key* using Advanced Encryption Standard (AES) in Electronic Code Book (ECB) mode. After one rotation step, the current counter, or *Ratchet*, is incremented. This is done until the local *Ratchet* is equal to the one that was received in the previous *Hint* message. Once the rotation is done the *SIV Key* is derived from the *Accessory Key*. This derivation is achieved by encrypting the static 32 byte string "`bt_aessivauthentbt_aessivencrypt`" with the *Accessory Key* using AES in ECB mode. Once the key derivation is finished, an *AES SIV* value is created. For this, the device creates a local random value, concatenates it with the received nonce and its own Bluetooth Address, and encrypts it with the *SIV Key*. This time, AES is used in SIV mode without a nonce or any additional data. A pseudo code representation of the ratcheting phase is shown in Listing 6.2.

```
# re-encrypt the accKey until ratchet values match
while local_ratchet < ratchet:
    accKey = aes_ecb(accKey, [0]*0x10)

# derive the siv_key
sivKey = aes_ecb(accKey, "btaessivauthentbt_aessivencrypt")

# create the AES SIV value by encrypting the received random value, the generated nonce
# as well as the remote device's Bluetooth address
AES_SIV = aes_siv(sivKey, remote_rand + host_nonce + remote_bd_addr)
```

Listing 6.2: Pseudocode of *MagicPairing* Ratcheting

At the end of this phase a *Ratchet AES SIV* message is sent back to the initiating device. It contains the local *Ratchet* value, as well as the *AES SIV* value. After receiving the message, the initiating device executes the same key derivation steps as mentioned above using the received *Ratchet* value. This leads to both devices having the same *Accessory Key* and *SIV Key*. Using the derived *SIV Key*, the initiating device can now decrypt the *AES SIV* value to unpack the random value of the responding device. If this decryption fails, the *MagicPairing* protocol is aborted and the receiver sends a *Status* message indicating *Failed Verification* (`0x07`).

PHASE 4: AES SIV PHASE

The initiating device will now create another *AES SIV* value. However, this one is different from the one that the *AirPods* created before. First, the device will create its own random value. Then it concatenates its nonce value, its random value, the *AirPods'* random value (which was received in the previous step) and the hint value. This 64 byte value is then encrypted with the derived *SIV Key* using AES in SIV mode and sent to the *AirPods*. If

the responding device can successfully decrypt the received data, it sends a *MagicPairing Status* message indicating success (`0x00`).

PHASE 5: LINK KEY DERIVATION PHASE

Now that the protocol messages are successfully exchanged, the Bluetooth link key is derived. To derive it, two *Session Pre Keys* are created and XORed. The *Session Key Pre 1* is created by encrypting the *AirPods'* random value with the initiating device's random value as key using AES in ECB mode. The second *Session Key Pre* is created by encrypting a 16 byte null-byte buffer with the responding device's random value using AES. Listing 6.3 shows a pseudocode representation of the link key derivation.

```
session_key_pre1 = aes_ecb(local_random, remote_random)
session_key_pre2 = aes_ecb(remote_random, [0] * 16)

# create the link key by XORing individual bytes of the pre keys
while i != 16:
    link_key[i] = session_key_pre1[i] ^ session_key_pre2[i]
```

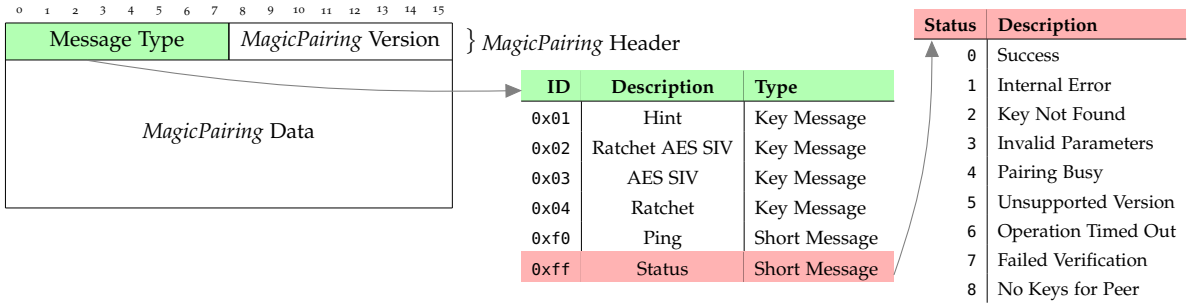
Listing 6.3: Pseudocode of *MagicPairing* Link Key Derivation

It is important to note that even though *Apple* implements a custom key derivation protocol for *MagicPairing*, it is still compliant to the Bluetooth standard and does not require any modifications to the Bluetooth firmware. link keys are stored on the host. During a standard Bluetooth pairing, the Bluetooth chip sends an Host Controller Interface (HCI) command to the host asking for the stored link key. In case of *MagicPairing*, the link key is not stored but derived. Instead of retrieving the link key from a local database, the host will instead derive the link key using *MagicPairing*. This behavior is completely transparent to the Bluetooth chip.

6.1.2 *MagicPairing Messages*

In the following, we will analyze the protocol messages that are exchanged during the *MagicPairing* phases. The general layout of a *MagicPairing* message looks as shown in Figure 17a. It starts with a two byte header, which is followed by data, depending on the type of the message. In general there are two different types of messages with a slightly different structure. The first type is a message that contains key material (such as the AES SIV, ratchet, or hint data), which we call *Key Message*. The second type contains just one byte of data after the header. The data in the key material messages is in a Type-Length-Value (TLV) structure. The number of keys is encoded directly after the header. The other message types, which we will further call *Short Messages*, directly contain a fixed amount of data after the header. The possible message types and possible key types are shown in Figure 17a.

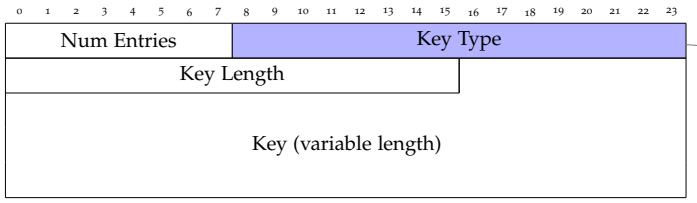
The *Ping* message is used to initiate the *MagicPairing* protocol. When a device receives a *MagicPairing* message, it replies with a Hint message. While the Ping message does not necessarily need any additional data, it is still three bytes long with the data set to `0x00`. The *Status* message is used to indicate success or, in case of an error, the reason for failing.



ID	Description	Type
0x01	Hint	Key Message
0x02	Ratchet AES SIV	Key Message
0x03	AES SIV	Key Message
0x04	Ratchet	Key Message
0xf0	Ping	Short Message
0xff	Status	Short Message

Status	Description
0	Success
1	Internal Error
2	Key Not Found
3	Invalid Parameters
4	Pairing Busy
5	Unsupported Version
6	Operation Timed Out
7	Failed Verification
8	No Keys for Peer

(a) *MagicPairing* message format



Key Type	Description
0x0010	Hint
0x0020	Nonce
0x0040	*Unknown*
0x0080	AES SIV
0x0100	Ratchet

(b) *MagicPairing* key message data

Figure 17: *MagicPairing* Packet Formats

Possible values for the data byte are shown in Figure 17b. Additionally, the *Ratchet* type message seems to be currently unused, as its reception handler implementation is empty on *iOS* and *macOS bluetoothd*.

6.1.3 *ProximityPairing Advertisements*

One of the triggers that initializes a connection backed by *MagicPairing* are *ProximityPairing* Bluetooth Low Energy (BLE) advertisements. They are sent out by *AirPods* when their case is opened or when they are taken our of their case. These advertisements are used to notify other *Apple* devices of the presence of the *AirPods*. Additionally, information such as the *AirPods* battery state, their color, and even a counter that counts the times the case was opened are encoded in these advertisements [32]. When an *iOS* device receives such an advertisement for a pair of *AirPods* that belong to the same *Apple* ID as the *iOS* device, a pop-up with an image of *AirPods*, the name of the *AirPods* and the current battery state is shown.

The advertisement data is divided into a plaintext part and a cipher text part. The latter part constitutes the encrypted *MagicPairing* data. The encrypted data is encrypted similarly to other *MagicPairing* data using AES in ECB mode. In addition to the previously shown *Accessory* keys, a new key is introduced, that is simply called *EncryptionKey*. The plaintext part of the *ProximityPairing* advertisements has been analyzed in previous work [32].

In addition to the *ProximityPairing* advertisements that have been documented by Martin et al. in [32], we found that there is another type of *ProximityPairing* advertisements. When the button on the *AirPods* case is pressed until the LED on the case is flashing

white, the *AirPods* are in pairing mode. When this mode is triggered, they emit an additional type of BLE advertisements. The beginning of the data is similar to the already documented *ProximityPairing* advertisements. The main difference is that the advertisement data contains the Bluetooth address of the *AirPods*. The third field, which was previously documented as the fixed value `0x00` is `0x01` here. Thus, we assume that this field indicates the type of *ProximityPairing* advertisement. The structure of these advertisements is shown in Figure 18. The Bluetooth address is followed by six other bits which we did not reverse engineer any further.

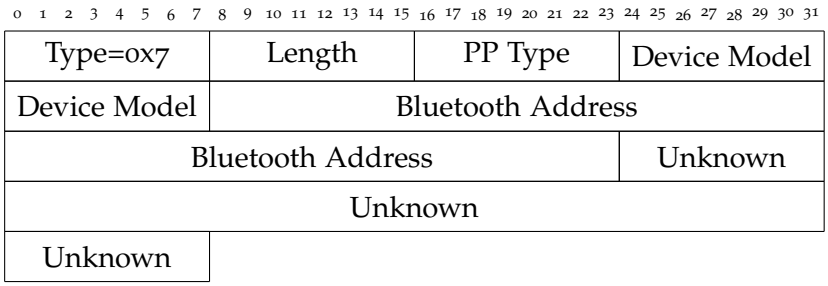


Figure 18: *AirPods ProximityPairing* Public Advertisement

6.2 ANALYZING THE MAGICPAIRING PROTOCOL

The *MagicPairing* protocol displays an interesting target for multiple reasons. Firstly, it offers an attacker the possibility to communicate using higher-level protocols (i.e., L2CAP) without being authenticated or paired to the victim. Secondly, there exist multiple different implementations that each might have their own flaws. The implementation used in *macOS* is written in Objective C, the implementation used in *iOS* (and its derivatives) is based on C/C++, and the implementation in the *AirPods* firmware is a slightly restricted variant also written in C. Lastly, the *MagicPairing* protocol is available on all *Apple* devices that have the ability to pair with *AirPods* regardless of whether the user owns them or not.

6.3 VULNERABILITIES IN THE MAGICPAIRING IMPLEMENTATIONS

In the following, the identified vulnerabilities in the *MagicPairing* protocol are described. Table 9 shows an overview of these vulnerabilities and their disclosure dates. Additional information about the execution of these vulnerabilities, such as the payloads, are provided in Appendix A.1.

Testing the *MagicPairing* protocol resulted in multiple NULL pointer dereferences, or dereferencing addresses in the NULL page, which is not mapped on 64 bit *iOS* and *macOS*. This results in a crash of the Bluetooth daemon. As it is managed by *launchd*, it will be restarted immediately after crashing. Thus, the bugs are merely a Denial of Service (DoS) of the Bluetooth daemon. An attacker does not have any control over the dereferenced value whatsoever, so the bugs are very likely not exploitable beyond a DoS.

ID	Attack	Effect	Detection Method	OS	Disclosure	Status
MP1	Ratchet AES SIV	Crash	Over-the-Air, In-Process	*iOS*	Oct 30 2019	Not fixed
MP2	Hint	Crash	Over-the-Air, In-Process	*iOS*	Dec 4 2019	Not fixed
MP3	Ratchet AES SIV	Crash	Over-the-Air	*macOS*	Oct 30 2019	Not fixed
MP4	Hint	Crash	Over-the-Air	*macOS*	Oct 30 2019	Not fixed
MP5	Ratchet AES SIV	Crash	In-Process	*iOS*	Mar 13 2019	Not fixed
MP6	Ratchet AES SIV Abort	Crash	In-Process	*iOS*	Mar 13 2019	Not fixed
MP7	Ratcheting Loop	100 % CPU Load	Over-the-Air	*macOS*	Oct 30 2019	Not fixed
MP8	Pairing Lockout	Disassociation	Manual	*iOS* & *macOS*	Feb 16 2020	Not fixed

Table 9: List of Identified *MagicPairing* Vulnerabilities

6.3.1 *MP1: iOS MagicPairing Ratchet AES SIV*

When sending a *MagicPairing Ping* message to an *iOS* device from a Bluetooth device that is not known as a pair of *AirPods* to the *iOS* device, it responds with the status message that it does not have a hint for this sending device (`0x08`). If a *Ratchet AES SIV* message is then sent to the device, *bluetoothd* will crash while trying to dereference a pointer in the NULL page. Listing 6.4 shows an excerpt of the crash log that is generated by the operating system.

```
Exception Type:  EXC_BAD_ACCESS (SIGSEGV)
Exception Subtype: KERN_INVALID_ADDRESS at 0x00000000000000a8
VM Region Info: 0xa8 is not in any region.  Bytes before following region: 4298293080
[...]
Termination Signal: Segmentation fault: 11
Termination Reason: Namespace SIGNAL, Code 0xb
Terminating Process: exc handler [958]
Triggered by Thread:  2
```

Listing 6.4: Excerpt of a *MagicPairing* Related Crash Log

The reason for this invalid access to the address `0xa8` is a missing check for the return value of a lookup function. The function looks up an entry in *bluetoothd*'s table of known *MagicPairing* devices by the sender's Bluetooth address. If there is no entry for this specific address, the function returns `NULL`. The issue is that this return value is never checked and assumed to be a pointer to a valid *MagicPairing* related structure. Then, to respond to the *Ratchet AES SIV* message, the structure is accessed at offset `0xa8`, which leads to the crash.

```
void receive_magic_pairing_ratchet_aes_siv(char *bd_addr, char *data) {
    [...]
    // as the Bluetooth address is not known as a magic paired device, this
    // function will return NULL
    magic_pairing_entry = lookup_mp_entry_by_bd_addr(bd_addr);
    [...]
    // the entry will still be dereferenced (with an offset) here
    memmove(magic_pairing_entry->remoteAESSIV, data + aessiv_offset, 0x36);
}
```

Listing 6.5: Pseudocode of a *MagicPairing* Pointer Dereference Vulnerability

6.3.2 *MP2–5: MagicPairing Hint and Ratchet AES SIV*

These vulnerabilities have the same cause as the previous one. The return value of the lookup function is not properly verified. On *macOS* this also affects the *Hint* and the *Ratchet AES SIV* messages. As before, both lead to a dereference of an invalid address, which is a fixed offset into a *MagicPairing* structure at address `0x0`. Thus, both vulnerabilities are equally unlikely exploitable other than crashing *bluetoothd*.

6.3.3 *MP6: Ratcheting Abort*

This crash is caused by an assertion failure that leads to an abort. The code that parses the *Ratchet AES SIV* message attempts reading from the message buffer. An assertion ensures that it does not read beyond this buffer. However, if the assertion fails, the parser does not return gracefully and instead calls abort, which leads to the termination of *bluetoothd*.

6.3.4 *MP7: Ratcheting Loop DoS*

The ratcheting DoS vulnerability results in a DoS attack against the receiving thread of *bluetoothd* on *macOS*. Under very specific circumstances, *macOS*'s *bluetoothd* can be forced to enter a ratcheting loop with a very large iteration count. Unlike the previous vulnerabilities, this issue is not solely caused by implementation mistakes, but relies on an inherent problem in the protocol's design. The flaw lies in the fact that the receiver will trust the values sent in the *Hint* message, without verifying that it was actually sent by a known *MagicPairing* peer. An attacker can forge the *Ratchet* value in the *Hint* message. Setting the *Ratchet* to a very high value will cause *bluetoothd* to enter a long ratcheting loop. The *Ratchet* field holds a 4 byte value, thus the maximum value of a *Ratchet* can be `0xffffffff`. During normal usage however, the *Ratchet* is only incremented for every pairing process. Therefore, in practice it will be rather small. The attack was tested on a MacBook Pro Early 2015, 13-inch, 2.9 GHz dual core i5 on *macOS* Catalina 10.15 with an initial *Ratchet* value of 2. Sending a *Hint* message with a *Ratchet* value of `0xffffffff` caused *bluetooth* to enter a ratcheting loop, with the local *Ratchet* value increasing by about 7.000 per second. Which would result in a ratcheting loop of multiple days.
The result is that the Bluetooth reception thread is blocked, which disturbs further communication based on Bluetooth. The targeted device will, for example, lose the ability to receive files via the *AirDrop* feature.

6.3.5 *MP8: Pairing Lockout Attack*

It is possible to corrupt the established pairing between an *iOS* or *macOS* device and a pair of *AirPod*s. For this, an attacker needs to know the victim's Bluetooth address, as well as the targeted *AirPod*s' Bluetooth address. The attacker can manipulate the local ratchet value of a host device by sending one or more *Ratchet AES SIV* messages with a ratchet value higher than the device's current one. The current ratchet value can be obtained by sending a *Ping* message to the host. It will then respond with a *Hint* message, which contains its current local ratchet value. This value can then be incremented and sent in a *Ratchet AES SIV* message. The keys for encrypting the AES SIV value are not

required. The ratchet value is sent unencrypted in the message. Therefore, an attacker can set a bogus value for the AES SIV part of the message and set the incremented ratchet value. This will make the receiving host start a ratcheting loop. As *bluetoothd* on *iOS* has a timeout functionality, the forged ratchet value should not be chosen too high, otherwise the pairing process tends to time out. As soon as the ratcheting loop is finished, the host's local ratchet value is successfully increase, even if the decryption of the AES SIV entry of the message fails. The reason why this corrupts an active pairing is that the *AirPods* have a threshold value for the discrepancy between their local ratchet value and the value received by the paired host. A pseudocode representation of this check is shown in Listing 6.6.

```
1  if localRatchet < remoteRatchet - 9:
2      sendMPStatus(1)
3  else
4      ratchetingLoop()
```

Listing 6.6: Pseudocode of *AirPods* Ratcheting

This causes the *AirPods* to decline the continuation of the *MagicPairing* protocol and thus the whole pairing process. The user does not have any options to reset the *MagicPairing* data and does not get any feedback about the actual error. The only solution is to reset the *AirPods* and freshly pair them with the user's *iCloud* account.

The attack can be conducted as follows:

1. The attacker changes their Bluetooth address to that of the target's *AirPods*.
2. The attacker connects to the victim and sends a *Ping* message to the victim to initiate a *MagicPairing* process.
3. The victim responds with a *Hint* message which contains its current local ratchet value.
4. The attacker increases this value by 10 and sends a *Ratchet AES SIV* message with the incremented ratchet value and a random AES SIV value.
5. The victim will start the ratcheting loop with the received ratchet value and derive the SIV key for decrypting the AES SIV value. As the AES SIV value is random, the victim will not be able to decrypt it and sends a *Status* message indicating an internal error. However, its local ratchet value stays incremented and is not set back to its previous value.

An overview of the attack is shown in Figure 19.
The issue originates from the fact that an untrusted ratchet value is used to increment an internal value und execute a key rotation. As the ratchet value is neither encrypted nor authenticated, an attacker can easily forge this value. A solution to this problem can be to only store the incremented ratchet value and the rotated key when the AES SIV part of the message was successfully decrypted. Otherwise, the whole *MagicPairing* message should be considered untrusted and the ratchet value should stay as it was before.

Another interesting note is that the implementations of *MagicPairing* on *iOS* and *macOS*

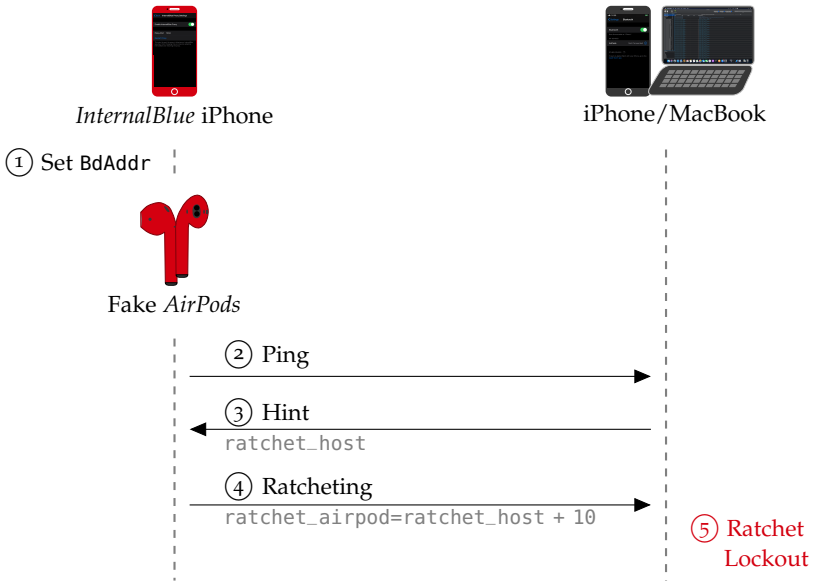


Figure 19: *MagicPairing* Lockout Attack

contains various spelling mistakes in logging messages, and in case of the *macOS* version of *bluetoothd* also in symbol names. While spelling mistakes are not directly related to flaws in an implementation, they do give the impression the code was not properly reviewed or else someone could have already found and corrected at least some of them. Our assumption likely proved right. The flaws we found regarding the NULL pointer dereferences are small coding mistakes that should have been caught during code review, similar to the spelling mistakes.

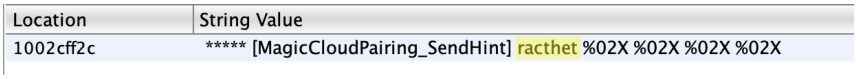
Location	String Value
1002c35f3	[MagicPairingManager] init and accountStatus and uplaod status : %d
1002aaa28	_uplaodStatus
1002c375d	deletedAllMagicCloudDevice : Uplaod status is false so do not delete %d
1002aa515	setUplaodStatus:
1002c3fff	Ti,V_uplaodStatus
1002aa508	uplaodStatus
1002c37f4	UplaodStatus
1002c3ff2	uplaodStatus

Figure 20: *macOS* 15.3 *bluetoothd* Upload Spelling

Location	String Value
1002d1ab2	***** [BT_AccountPairing_RacheteAccKey] acc_key: %02X %02X %02X %02X %0...
1002d1b35	***** [BT_AccountPairing_RacheteAccKey] new_acc_key: %02X %02X %02X %02...
1002cf960	***** [BT_AccountPairing_SendAesSIV] didRatchet racheted Key: %02X %02X %0...
1002cf9f0	***** [BT_AccountPairing_SendAesSIV] No ratchet rachetedKey: %02X %02X %0...
1002b07f9	BT_AccountPairing_RacheteAccKey:new_acc_key:
1002b0a09	rachetedKey

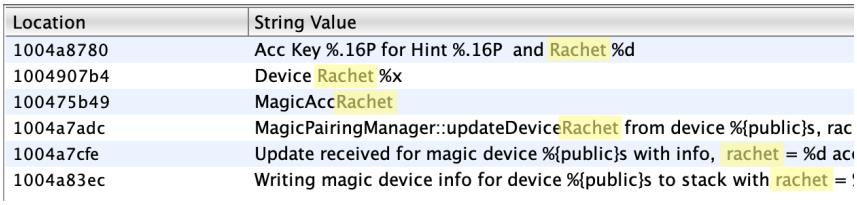
Figure 21: *macOS* 15.3 *bluetoothd* Ratchet Spelling

Given the attack surface of the protocol a thorough review should have certainly be done before releasing the code in production. Examples of such spelling mistakes are shown in Figures 20 to 23. Note that the word *ratchet*, has been spelled differently various times.



Location	String Value
1002cff2c	***** [MagicCloudPairing_SendHint] racthet %02X %02X %02X %02X

Figure 22: *macOS* 15.3 *bluetoothd* Ratchet Spelling



Location	String Value
1004a8780	Acc Key %.16P for Hint %.16P and Rachet %d
1004907b4	Device Rachet %x
100475b49	MagicAccRachet
1004a7adc	MagicPairingManager::updateDeviceRachet from device %{public}s, rac
1004a7cfe	Update received for magic device %{public}s with info, rachet = %d ac
1004a83ec	Writing magic device info for device %{public}s to stack with rachet =

Figure 23: *iOS* 13.3 *bluetoothd* Ratchet Spelling

We assume that the intention was to call it *ratchet*, which is supported by the fact that this is the spelling that is used most commonly (instead of *rachet* or *rachete*).

7 DISCUSSION AND FUTURE WORK

In this chapter, we discuss the results of this thesis. In Section 7.1, we begin by summarizing and discussing the vulnerabilities that were identified. Following that, we discuss the our *InternalBlue* proxy implementation in Section 7.2. Lastly, we describe improvements that can be made to our in-process fuzzer (see Section 7.3).

7.1 SUMMARY AND DISCUSSION OF IDENTIFIED VULNERABILITES

In total, we identified 13 issues and vulnerabilities using a combined approach of over-the-air fuzzing, in-process fuzzing, as well as manual reverse engineering. Table 10 shows an overview of these vulnerabilities along with the date they were reported and their current status.

ID	Description	Effect	Detection Method	OS	Disclosure	Status
MP1	Ratchet AES SIV	Crash	Over-the-Air, In-Process	iOS	Oct 30 2019	Not fixed
MP2	Hint	Crash	Over-the-Air, In-Process	iOS	Dec 4 2019	Not fixed
MP3	Ratchet AES SIV	Crash	Over-the-Air	macOS	Oct 30 2019	Not fixed
MP4	Hint	Crash	Over-the-Air	macOS	Oct 30 2019	Not fixed
MP5	Ratcheting Loop	100 % CPU Load	Over-the-Air	macOS	Oct 30 2019	Not fixed
MP6	Pairing Lockout	Disassociation	Manual	iOS & macOS	Feb 16 2020	Not fixed
MP7	Ratchet AES SIV	Crash	In-Process	iOS	Mar 13 2020	Not fixed
MP8	Ratchet AES SIV	Crash	In-Process	iOS	Mar 13 2020	Not fixed
L2CAP1	Fixed Channel	Crash	Over-the-Air	RTKit	Dec 4 2019	Not fixed
L2CAP2	Group Message	Crash	In-Process	iOS	Mar 13 2020	Not fixed
LEAP1	Version Leak	Information Disclosure	Manual	iOS	Mar 31 2020	Not fixed
SMP1	SMP OOB	Partial PC Control	In-Process	iOS	Mar 31 2020	Not fixed
SIG1	Missing Checks	Crash	In-Process	iOS	Mar 31 2020	Not fixed

Table 10: List of Total Identified Vulnerabilities

While we did not find any vulnerabilities that obviously lead to a remote compromise of the attacked *Apple* device, we identified multiple vulnerabilities that indicate a lack of quality assurance in *Apple*'s Bluetooth stacks. Both the spelling mistakes and the type of bugs in the *MagicPairing* protocol indicate a severe lack of code reviews and testing. This impression is supported by *L2CAP2*, which has existed since at least 2011. Furthermore, the protocols we tested seem to be of varying quality. The *MagicPairing* implementations showed some very shallow bugs, while the Generic Attribute (GATT) protocol, seems much more solid. As of now, we did not find any flaws within this protocol using our fuzzing framework *ToothPicker*.

Additionally, we identified a flaw in both the *Magnet* and *LEAP* protocols that disclose the device's software and hardware versions. Both of these protocols are available via Bluetooth Low Energy (BLE). Therefore, an attacker can connect to a randomized BLE

address and use the version leak to leak both the hardware and the software version of the device. Another, more drastical vulnerability was found in the Security Manager Protocol (SMP) accessible via both BLE and Classic Bluetooth. In the Classic Bluetooth variant, it was possible to gain partial control over the program counter, which could potentially result in hijacking the control flow. Due to the high complexity and large amount of protocols that can be tested, we were only able to cover a subset of the available protocols within the scope of this thesis. We highly recommend a continuation of the testing efforts in both breadth (i.e., more protocol coverage) and depth, as our current findings indicate quality issues within the many of the protocol handlers already.

While *bluetoothd* is a user-mode daemon, it has high privileges. It can, for example access the phone's contacts, and communicate with other highly privileged daemons, such as *wifid*, which runs as the root user. A vulnerability leading to Remote Code Execution (RCE) in *bluetoothd* is thus highly critical. Devices with a *Marconi* Bluetooth chip or a Bluetooth chip connected via Peripheral Component Interconnect Express (PCIe) are even more critical, as the communication with the Bluetooth chip involves drivers in the kernel.

Some of these issues listed in Table 10 have been reported to *Apple* more than 90 days ago as of this writing. However, *Apple* did not reacted to the reports beyond acknowledging the reception of the reports and asking for more information and proof-of-concepts. While we understand that there are certainly more severe vulnerabilities that need to be fixed within *Apple*'s large ecosystem, we were surprised that our reports have been mostly disregarded. None of the reported issues have been resolved and the status of the submissions is completely unknown to us.

7.2 INTERNALBLUE ON IOS

In Section 4.1, we have shown how *InternalBlue* was ported to *iOS*. The solution currently works on *iOS* devices with a Universal Asynchronous Receiver Transmitter (UART) Bluetooth chip, which was sufficient at the beginning of this thesis. There was no publicly available *Jailbreak* for the newer devices that have a Bluetooth chip connected via PCIe, which is required to develop such invasive tools like the *InternalBlue* proxy daemon. Therefore, we only theoretically evaluated how the daemon can be implemented for devices with PCIe Bluetooth. As there now are public *Jailbreaks* available for these newer devices, the implementation can be done in future work. In the following, we briefly outline how the proxy can be implemented with these PCIe devices.

Both *BlueTool* and *bluetoothd* use the `AppleConvergedTransport.dylib` library to connect to the PCIe Bluetooth chip. This library provides a simple interface and wraps PCIe-specific code. There are four important functions in this library that are required for creating the *InternalBlue* proxy daemon:

- `AppleConvergedTransportInitParameters`
- `AppleCovergedTransportCreate`
- `AppleConvergedTransportWrite`

- `AppleConvergedTransportFree`

First, the `AppleConvergedTransportInitParameters` function is used to initialize a structure that will later be passed as first argument to `AppleCovergedTransportCreate`. The second argument to this function is a pointer to the transport handle, which is later required as a reference to this specific transport. Different to the UART-based connection, the different protocols Host Controller Interface (HCI), Asynchronous Connection-Less (ACL), and Synchronous Connection-Oriented (SCO) require separate transports. Thus, the `AppleCovergedTransportCreate` function needs to be called for each of these. The transport structure argument determines the type of transport. In addition to that, it also defines a callback function that is called whenever data is received at this transport. This incoming data has to be forwarded to *InternalBlue*. Data that is received from *InternalBlue* can be forwarded to the Bluetooth chip by calling the `AppleConvergedTransportWrite` function. Once the proxy is stopped, the transports can be closed by calling the function `AppleConvergedTransportFree` along with the transport handles.

7.3 FUZZING IMPROVEMENTS

In this thesis, we implemented an in-process fuzzer that can be used to fuzz protocol handler functions in user-space applications on *iOS* as part of our *ToothPicker* fuzzing framework. We used this approach to specifically fuzz a selection of Bluetooth protocols and were able to identify multiple bugs and vulnerabilities. During the operations, we found the potential for several improvements, where the implementation would be out of scope for this thesis. Yet, we still want to mention them for future work.

7.3.1 *Conversation-Oriented Fuzzing*

The current fuzzing approach is message-based. This means that each fuzzing payload that is delivered is generated and sent individually, disregarding the overall flow of messages. However, protocols, such as *Magnet* or the Logical Link Control and Adaptation Protocol (L2CAP) Signal Channel, have a state that might be altered depending on the received messages. Therefore, not only individual messages are important, but rather the entirety of all sent messages within a connection. We already implement a stripped-down conversation-oriented approach in the *Magnet* protocol's specialized harness (see Section 5.3). By storing the information about the current connection's protocol version, the harness *knows* which protocol version it currently has to use. Depending on this version, the messages are adapted. In the case of the *Magnet* protocol, this influences the size of the length field.

A conversation-oriented approach could be built in such a way that the corpus, as well as the mutated input, consists not only of a single message, but multiple messages. In general, this approach requires various modifications to both the fuzzing master and the harness. A limitation of this approach is that the fuzzer needs to do a *cleanup* after finishing a conversation. Depending on what state is considered, i.e., whether the whole memory of the program is considered as state, or just the actual protocol state, this cleanup is time-consuming. In case the whole memory is considered as state, *bluetoothd* has to be restarted after each conversation. If only the protocol's state is considered, the

virtual connection needs to be disconnected and reconnected. The conversation-oriented approach also has the advantage that potential deviations in the target's responses can be detected, which is an indicator for unexpected behavior.

Previous work, such as [11] and [1], implement a similar approach for networking protocols.

7.3.2 *Concurrent Fuzzing*

In the current implementation, we only create a single virtual connection to fuzz the reception handler functions. However, protocols like *Magnet* or *MagicPairing* hold global protocol state in both global variables and heap memory. Testing for flaws in the handling of concurrent connections requires the fuzzer to create and maintain multiple connections. Testing for flaws in these concurrent connections is relevant for security as the concurrency might influence the allocation of heap memory and handling of threats. A concurrency feature mainly requires adaptions in the fuzzing harness. It needs to create two or more virtual connections and then call the reception handlers with one of the created connections.

7.3.3 *Input Generation and Mutation*

The in-process fuzzer currently relies on *radamsa* [15] as an input mutation component. *Radamsa* is called by the fuzzing manager for each individual input form the corpus, which is a large performance overhead. A different input-generation strategy should be chosen to improve the fuzzer's performance. One possibility is to resort to well-established tools, such as *AFL* [46]. While the development of original *AFL* has stopped, the project is continued as *AFL++* by Heuse et al. [16]. In the branch called *frida* on the *AFL++* GitHub repository, the developers started to implement FЯIDA support for *AFL++*. An initial observation of this code implies that our in-process fuzzer can be adapted to *AFL++* and thus benefit from its features, such as the improved input mutation and generation.

Our assumption is that adapting our in-process fuzzer to work with *AFL++* will bring major improvements in performance and results for the following reasons:

- *AFL++* runs on the *iOS* device itself, which is much faster than a script running in *Python* on a host computer connected to the *iOS* device via Universal Serial Bus (USB).
- In consequence, the FЯIDA runtime can be used directly by the fuzzer by using the *frida-core* library for injecting the FЯIDA scripts [9]. This effectively eliminates the overhead of the USB connection, and the FЯIDA server listening on TCP on the *iOS* device.
- Lastly, the fuzzer would benefit from improved input generation mechanisms, such as the recently introduced MOPT [30].

Implementing this combined fuzzer requires a large engineering effort on top of the fuzzer we already built. Thus, this implementation was out of scope for this thesis but might be considered in future work.

7.3.3.1 *Sanitizers*

The current implementation of the fuzzer can reliably detect bugs that lead to a crash, e.g., when unmapped memory is accessed. However, it cannot detect bugs that lead to minor deviations, such as buffer overflows that lead to leaking data in protocol responses, or overwrite other non-vital data in the program's memory. For such cases, tools like *Address Sanitizer* [38] are used. Employing *Address Sanitizer* on targets where no source code is available is generally more difficult. In these cases, strategies like dynamic instrumentation or *AFL*s *libdislocator* [10] are used. *Apple* provides developers using *Xcode* the ability to enable *Address Sanitizer* for their own apps. Unfortunately, there is no binary-only *Address Sanitizer* solution for *iOS*, yet. FЯIDA's binary instrumentation capabilities can be used to instrument memory allocation and deallocation routines to detect possible overflows. However, the implementation of such an instrumentation was out of scope for this thesis. Nonetheless, this is highly encouraged as future work due to the additional bug classes that can be detected using this approach.

8 CONCLUSIONS

In this thesis, we provided an overview over *Apple*'s proprietary Bluetooth stacks with a focus on the *iOS* operating system. We uncovered multiple custom Bluetooth protocols built by *Apple* and briefly documented a subset of them. We chose one of these protocols, the *MagicPairing* protocol, which we reverse-engineered and documented in detail. Furthermore, we implemented *ToothPicker*, a framework consisting of an over-the-air and an in-process fuzzer to target Bluetooth protocols on *iOS*. By a combination of fuzzing and manual analysis we uncovered multiple flaws in the set of chosen protocols.

We briefly introduced related work and the Bluetooth technology in Chapter 2. In Chapter 3, we provided a general overview over *Apple*'s three distinct Bluetooth stacks on *macOS*, *iOS*, and *RTKit*. Providing insight about the architecture, the features, and the involved components serves as a basis for the following security research. It also aids any future work on Bluetooth in the *Apple* ecosystem. We discovered that *Apple* developed, and thus has to maintain, three completely different Bluetooth stacks for their different operating systems. Moreover, the architecture of these stacks have only little in common. We specifically picked the *iOS* Bluetooth stack which we analyzed and explained in more detail. Afterwards, we provided an overview of some of the proprietary protocols *Apple* integrated into their Bluetooth ecosystem. It turns out that *Apple* does not only integrate additional protocols that augment the Bluetooth standard, such as the *MagicPairing* protocol, but also deviates from it, for example with the *Fast Connect Discovery* protocol. In Section 3.3, we documented the distribution of *Apple*'s real-time operating system in Bluetooth peripherals. We also presented the discovery of *Apple*'s very own Bluetooth chip *Marconi*.

Based on a preliminary analysis, we provided a list of protocols that are interesting for security research and prioritized them for further analysis (see Section 3.5). The prioritization in Table 5 shows that there is a huge attack surface which is impossible to fully cover within the scope of a thesis. Following that, we presented our manual testing methodology in Chapter 4. This includes both static analysis, such as reverse-engineering Bluetooth-related components, and dynamic analysis, such as capturing Bluetooth traffic. We reverse-engineered several of *Apple*'s Bluetooth-related daemons and frameworks. Again, the amount of possible targets is huge and achieving full coverage is hardly possible.

In addition to the manual testing efforts, we implemented *ToothPicker*, a fuzzing framework that includes an over-the-air fuzzer, as well as an in-process fuzzer. The over-the-air fuzzer can target any Bluetooth stack. It is implemented on the Bluetooth research platform *InternalBlue*. As its effectiveness and speed are constrained, we also implemented a FЯIDA-based in-process fuzzer for Bluetooth protocol handlers on the *iOS* operating

system. It is generic enough to target different Bluetooth protocols and protocol handlers on *iOS* and thus enables further research beyond this work.

With the combination of the manual and automated fuzzing approaches we found 13 vulnerabilities in total, covering all three Bluetooth stacks. Lastly, we analyze and document the *MagicPairing* protocol thoroughly. While we found that all of the three individual implementations are flawed, the general idea and concept of the protocol is promising. It provides a seamless pairing capability for cloud-connected devices, while guaranteeing a fresh Bluetooth link key on each pairing. On top of that, the protocol is compliant with the Bluetooth standard. We also think that other vendors and IoT ecosystems could benefit from this protocol and its infrastructure. All *Apple* needs to do is to provide developers an Application Programming Interface (API) to generate and synchronize an *Accessory Key* with the user's *iCloud* account. Vendors implementing the *MagicPairing* protocol could then benefit from the seamless pairing mechanism currently reserved for the *AirPods*.

Parts of *Apple*'s ecosystem are a closed black box. The Bluetooth stack is no exception. This thesis serves as an attempt to shed some light into this black box, get a first impression of its security, and pave the way for future research. We were able to identify multiple flaws within the implementations of both the proprietary Bluetooth protocols and the protocols specified in the Bluetooth standard. Many of these give the impression that the *iOS* Bluetooth stack lacks thorough code reviews. The protocols we targeted are always available as long as Bluetooth is enabled. The Bluetooth Low Energy (BLE) protocols are even possible to reach with no prior knowledge of the Bluetooth address of the victim. These are great preconditions for zero-click Remote Code Execution (RCE) attacks. Thus, we would have assumed that *Apple* had more thorough code review and testing practices in place. We encourage further research of the *Apple* Bluetooth stacks with a special focus on the proprietary protocols.

A

APPENDIX

A.1 PROOF-OF-CONCEPT PAYLOADS

This section provides more details on the vulnerabilities that were found, such as the payload that needs to be sent and what precautionary measures have to be taken before sending the payload.

There are three types of prerequisites that might be required for certain proof-of-concepts, especially the for the ones targeting the *MagicPairing* protocol:

UNKNOWN BLUETOOTH ADDRESS: the attacker's Bluetooth address must not be known by the victim as a device paired with *MagicPairing*.

INCREASE MTU WITH PING: some of the payloads are longer than the initial L2CAP Maximum Transmission Unit (MTU). This can be overcome by sending a *MagicPairing Ping* message[1]. Alternatively, this can be achieved by specifying the MTU in Logical Link Control and Adaptation Protocol (L2CAP) *Information Requests*. However, the approaching using the *Ping* message is much faster.

KNOWN AIRPODS ADDRESS: the attacker's Bluetooth address needs to correspond to the Bluetooth address of a pair of *AirPods* that are known and paired with the victim.

A.1.1 *MagicPairing*

[MP1] iOS RatchetAESSIV Crash	
Result	Pointer dereference crash at `0xa8`
Technology	Classic Bluetooth
Prerequisites	- unknown Bluetooth address - increase MTU with Ping
L2CAP CID	`0x30`

L2CAP Payload	`02010280003600AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA`
	`AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA`
	`AAAAAAAAAAA001040012345678`

1 The *MagicPairing Ping* message can be transmitted by sending the following L2CAP payload to CID `0x30`: `F00000`.

[MP2] iOS Hint Crash	
Result	Pointer dereference crash at `0x01`
Technology	Classic Bluetooth
Prerequisites	- unknown Bluetooth address
L2CAP CID	`0x30`
L2CAP Payload	`01020304050607`

[MP3] macOS RatchetAESSIV Crash	
Result	Pointer dereference crash at `0x00`
Technology	Classic Bluetooth
Prerequisites	- unknown Bluetooth address - increase MTU with Ping
L2CAP CID	`0x30`

L2CAP Payload	`02010280003600AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA`
	`AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA`
	`AAAAAAAAAAA001040012345678`

[MP4] macOS Hint Crash	
Result	Pointer dereference crash at `0x00`
Technology	Classic Bluetooth
Prerequisites	- unknown Bluetooth address - increase MTU with Ping
L2CAP CID	`0x30`
L2CAP Payload	`01010310001000AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA20001000BB` `BBBBBBBBBBBBBBBBBBBBBBBBBBBBBB0001040012345678`

[MP5] iOS RatchetAESSIV Crash	
Result	Pointer dereference crash at `0x10d`
Technology	Classic Bluetooth
Prerequisites	- unknown Bluetooth address - increase MTU with Ping
L2CAP CID	`0x30`

L2CAP Payload	`02010b028000360091b51d14747835f3a0818f7de4434329b3d4e265`
	`e5005b3f3ad5fdcaea6991b51d147478307de4434329b3d4e265e500`
	`5b3f3ad5fdcaea6991b51d14747834323934393637323935 7de44343`
	`29b3d4e265e5005b3f3ad5fdcaea6991a5580267a9a761bf4b046cf3`
	`0e4f6147a1a06bb74b5702d6c033343032383233363639323039333 8`
	`343633343633333734363037343331373638f3a081b4323131343831`
	`6c010104002b0100`

[MP6] iOS RatchetAESSIV Crash	
Result	Assertion failure crash
Technology	Classic Bluetooth
Prerequisites	- unknown Bluetooth address - increase MTU with Ping
L2CAP CID	`0x30`

L2CAP Payload	`02f3a081ae80002d330091b51d147478360104002b010000a393d231`
	`31fe617878f69af4207d34323934393637333033e22775642f7fc1cd`
	`9fdcddc89934dd39608afc6948b87ee0ef8968286341fd0515f98acd`
	`5fb62f55f923887021a4ea8730cbaae05058b60f673c510a6170aa2e`
	`cbdf1d142f763ef03f38d27c392ecdf1a574fdf906bcf74aa35da085`
	`f137ddecff2aec0d5c95b8fa83a71b42af205359e4f02aaca2ab4778`
	`001274a818333430323832333636393230393338343633343633333 7`
	`34363037343331373638323131343536057f`

[MP7] macOS RatchetAESSIV DoS	
Result	Ratcheting DoS
Technology	Classic Bluetooth
Prerequisites	- set BD address to known AirPods - increase MTU with Ping
L2CAP CID	`0x30`

L2CAP Payload	`020102800036 00AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA`
	`AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA`
	`AAAAAAAAAA00010400fffffff0`

[MP8] AirPods Lockout
See Section 6.3.5 for a detailed description of this attack.

A.1.2 *L2CAP*

[L2CAP1] AirPods L2CAP Crash	
Result	Unspecified crash
Technology	Classic Bluetooth
Prerequisites	-
ACL Payload	`0b20040000003000`

[L2CAP2] iOS L2CAP Group Crash	
Result	NULL-pointer jump
Technology	Classic Bluetooth / BLE
Prerequisites	-
L2CAP CID	`0x02`
L2CAP Payload	`000001000200`

A.1.3 *LEAP*

[LEAP1] iOS LEAP Version Info	
Result	Hardware and Software Version Leak
Technology	BLE
Prerequisites	-
L2CAP CID	`0x2A`
L2CAP Payload	`014C45415001004C0013001100`

A.1.4 *SMP*

[SMP1] iOS SMP	
Result	Opcode handler OOB read + jump
Technology	Classic Bluetooth / BLE
Prerequisites	-
L2CAP CID	`0x06` (BLE) and `0x07` (Classic)
L2CAP Payload	`0f414243444546474849`

A.1.5 *L2CAP Signal Channel*

[SIG1] iOS Signal Channel Crashes	
Result	Invalid pointer dereferences
Technology	Classic Bluetooth
Prerequisites	-
L2CAP CID	`0x01`
L2CAP Payload `(0x00)`	`04000c0007000100070a00070a0400070a040009323535040307 0b`
L2CAP Payload `(0x08)`	`040508000c0001808000060006000600060006000606000605080 00f3b003bff`
L2CAP Payload `(0x10)`	`0700010007000400072904000729040007290400072904 32`

A.2 FUNCTION OFFSETS ON IOS 13.3

This table lists the offsets of the functions that are used in the fuzzing harness of *ToothPicker*. These addresses are only valid for *bluetoothd* on *iOS* 13.3. However, they can be used to deduct the function offsets of other versions for future research.

Function Name	**Offset Address**
`allocateACLConnection`	`0xc81a0`
`allocateLEConnection`	`0x11ba24`
`LE_ReadRemoteVersionInformationComplete`	`0x11c948`
`ReadRemoteVersionInformationCB`	`0x12117c`
`getConnectionStruct`	`0x0c7df4`
`OI_HCI_ReleaseConnection`	`0x0c87e0`
`OI_LP_ConnectionAdded`	`0xf02b8`
`OI_SignalMan_Recv`	`0x10b52c`
`OI_L2CAP_Recv`	`0x1031ec`
`magnet_l2cap_recv`	`0x403d8`
`magic_pairing_handler`	`0x129110`
`ACL_reception_handler`	`0xcd824`

BIBLIOGRAPHY

[1] Kaled Alshmrany and Lucas Cordeiro. "Finding Security Vulnerabilities in Network Protocol Implementations." In: *arXiv preprint arXiv:2001.09592* (2020).

[2] Apple. *Bug Reporting—Profiles and Logs*. https://developer.apple.com/bug-reporting/profiles-and-logs/. 2020. (Visited on 03/01/2020).

[3] Bluetooth SIG. *Bluetooth Core Specification 5.2*. https://www.bluetooth.com/specifications/bluetooth-core-specification. Jan. 2020.

[4] Guillaume Celosia and Mathieu Cunche. "Discontinued Privacy: Personal Data Leaks in Apple Bluetooth-Low-Energy Continuity Protocols." In: *Proceedings on Privacy Enhancing Technologies* 2020.1 (2020), pp. 26–46.

[5] Stefan Esser. *Introduction to the Security of Apple Pencil and Apple Smart Keyboard*. Tech. rep. 2018. URL: https://gsec.hitb.org/materials/sg2018/WHITEPAPERS/A%20First%20Look%20Into%20The%20Security%20of%20The%20Apple%20Pencil%20and%20the%20Apple%20SmartKeyboard%20-%20Stefan%20Esser.pdf.

[6] Stefan Esser. "Is the Pen mightier than the Sword?" Hack in the Box GSEC. 2018. URL: https://gsec.hitb.org/materials/sg2018/D1%20-%20A%20First%20Look%20Into%20The%20Security%20of%20The%20Apple%20Pencil%20and%20the%20Apple%20SmartKeyboard%20-%20Stefan%20Esser.pdf.

[7] *Frida - A world-class dynamic instrumentation framework*. URL: https://frida.re/ (visited on 02/24/2020).

[8] *Frida-based general purpose fuzzer*. 2019. URL: https://github.com/demantz/frizzer (visited on 02/24/2020).

[9] *Frida core library intended for static linking into bindings*. URL: https://github.com/frida/frida-core (visited on 03/27/2020).

[10] *GitHub AFL/libdislocator*. URL: https://github.com/google/AFL/tree/master/libdislocator (visited on 03/27/2020).

[11] Serge Gorbunov and Arnold Rosenbloom. "Autofuzz: Automated network protocol fuzzing framework." In: *IJCSNS* 10.8 (2010), p. 239.

[12] *Hacker News*. URL: https://news.ycombinator.com/item?id=20885719 (visited on 11/05/2019).

[13] D. Harkins. *Synthetic Initialization Vector (SIV) Authenticated Encryption Using the Advanced Encryption Standard (AES)*. RFC 5297. RFC Editor, 2008. URL: https://tools.ietf.org/rfc5297.txt.

[14] Alexander Heinrich. *Analyzing Apple's Private Wireless Communication Protocols with a Focus on Security and Privacy*. Master's Thesis. Nov. 2019.

[15] Aki Helin. *radamsa - a general-purpose fuzzer*. URL: https://gitlab.com/akihe/radamsa (visited on 02/24/2020).

[16] Marc Heuse, Heiko Eißfeldt, Andrea Fioraldi, and Dominik Maier. *The AFL++ fuzzing framework | AFLplusplus*. 2020. URL: https://aflplus.plus/ (visited on 03/27/2020).

[17] *IMG4 File Format - The iPhone Wiki*. URL: https://www.theiphonewiki.com/wiki/IMG4_File_Format (visited on 03/24/2020).

[18] Apple Inc. *Mach Overview*. 2013. URL: https://developer.apple.com/library/archive/documentation/Darwin/Conceptual/KernelProgramming/Mach/Mach.html (visited on 03/29/2020).

[19] Apple Inc. *Accessory Design Guidelines for Apple Devices*. Nov. 2019. URL: https://developer.apple.com/accessories/Accessory-Design-Guidelines.pdf (visited on 02/18/2020).

[20] Apple Inc. *Apple Security Bounty - Example Payloads*. 2020. URL: https://developer.apple.com/security-bounty/payouts/ (visited on 02/12/2020).

[21] Apple Inc. *XPC | Apple Developer Documentation*. 2020. URL: https://developer.apple.com/documentation/xpc (visited on 03/29/2020).

[22] Apple Inc. *CoreBluetooth | Apple Developer Documentation*. 202. URL: https://developer.apple.com/documentation/corebluetooth?language=objc (visited on 02/18/2020).

[23] Bluetooth SIG Inc. *Bluetooth Market Update 2019*. 2019. URL: https://3pl46c46ctx02p7rzdsvsg21-wpengine.netdna-ssl.com/wp-content/uploads/2018/04/2019-Bluetooth-Market-Update.pdf (visited on 11/14/2019).

[24] *Jailbreak - The iPhone Wiki*. 2020. URL: https://www.theiphonewiki.com/wiki/Jailbreak (visited on 03/30/2020).

[25] Counterpoint Research (Liz Lee). *Global True Wireless Hearables Market Reaches 12.5 Million Units in Q4 2018*. 2019. URL: https://www.counterpointresearch.com/global-true-wireless-hearables-market-reaches-12-5-million-units-q4-2018/ (visited on 10/23/2019).

[26] Jonathan Levin. *Spitting Image: MacOS & *OS Software Images*. Tech. rep. URL: http://newosxbook.com/bonus/vol1AppA.html (visited on 11/05/2019).

[27] Jonathan Levin. *Handling low memory conditions in iOS and Mavericks*. Tech. rep. Oct. 2016. URL: http://newosxbook.com/articles/MemoryPressure.html (visited on 03/10/2020).

[28] Jonathan Levin. **iOS Internals::User Space, Volume I (v1.1)*. 2018. URL: http://newosxbook.com.

[29] Jonathan Levin. **iOS Internals::Kernel Mode, Volume II*. 2019. URL: http://newosxbook.com.

[30] Chenyang Lyu, Shouling Ji, Chao Zhang, Yuwei Li, Wei-Han Lee, Yu Song, and Raheem Beyah. "MOPT: Optimized Mutation Scheduling for Fuzzers." In: *28th USENIX Security Symposium (USENIX Security 19)*. 2019, pp. 1949–1966.

[31] 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*. ACM. 2019, pp. 79–90.

[32] Jeremy Martin, Douglas Alpuche, Kristina Bodeman, Lamont Brown, Ellis Fenske, Lucas Foppe, Travis Mayberry, Erik Rye, Brandon Sipes, and Sam Teplov. "Handoff All Your Privacy–A Review of Apple's Bluetooth Low Energy Continuity Protocol." In: *Proceedings on Privacy Enhancing Technologies* 2019.4 (2019), pp. 34–53.

[33] Benjamin Mayo. *Apple rejecting Electron apps from Mac App Store due to private API usage*. Nov. 2019. URL: https://9to5mac.com/2019/11/04/electron-app-rejections/ (visited on 03/24/2020).

[34] *OTA Updates- The iPhone Wiki*. URL: https://www.theiphonewiki.com/wiki/OTA_Updates (visited on 03/08/2020).

[35] Jack Purcher. *A European Patent Filing from Apple Describes Voice Controls Destined for a Television and the Apple TV Set Top Box*. Nov. 2014. URL: https://www.patentlyapple.com/patently-apple/2014/11/a-european-patent-filing-from-apple-describes-voice-controls-destined-for-a-television-and-the-apple-tv-set-top-box.html (visited on 02/22/2020).

[36] *Research into Security of Apple Smart Keyboard and Apple Pencil*. URL: https://github.com/Antid0teCom/ipad_accessory_research (visited on 03/20/2020).

[37] Bluetooth SIG. *Bluetooth Core Specification 5*. Tech. rep. 2016. URL: https://www.bluetooth.com/specifications/bluetooth-core-specification (visited on 11/05/2019).

[38] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. "AddressSanitizer: A fast address sanity checker." In: *USENIX Annual Technical Conference 2012)*. 2012, pp. 309–318.

[39] Ben Seri and Gregory Vishnepolsky. "BlueBorne." In: (2017). URL: https://armis.com/blueborne.

[40] *Set up Wireless Audio Sync with Apple TV*. Nov. 2019. URL: https://support.apple.com/en-us/HT210526 (visited on 03/19/2020).

[41] Freddie Temperton. *Apple AirPod Charger Teardown and Reverse Engineering*. Jan. 2017. URL: https://mindtribe.com/2017/01/apple-airpod-charger-teardown-and-reverse-engineering/ (visited on 03/08/2020).

[42] Davide Toldo. "Analysing the macOS Bluetooth Stack." Bachelor's Thesis. Dec. 2019.

[43] *Twitter Shac Ron*. URL: https://twitter.com/stuntpants/status/1040705539833909248 (visited on 11/05/2019).

[44] *Usbmux - The iPhone Wiki*. URL: https://www.theiphonewiki.com/wiki/Usbmux (visited on 03/01/2020).

[45] The iPhone Wiki. *DCSD Cable*. 2016. URL: https://www.theiphonewiki.com/wiki/DCSD_Cable (visited on 10/23/2019).

[46] Michal Zalewski. *American fuzzy lop.(2015)*. 2015. URL: http://lcamtuf.coredump.cx/afl.