DEMO: Attaching InternalBlue to the Proprietary macOS IOBluetooth Framework

Davide Toldo
Secure Mobile Networking Lab
TU Darmstadt, Germany
dtoldo@seemoo.de

Jiska Classen
Secure Mobile Networking Lab
TU Darmstadt, Germany
jclassen@seemoo.de

Matthias Hollick
Secure Mobile Networking Lab
TU Darmstadt, Germany
mhollick@seemoo.de

ABSTRACT
In this demo, we provide an overview of the macOS Bluetooth stack internals and gain access to undocumented low-level interfaces. We leverage this knowledge to add macOS support to the InternalBlue firmware modification and wireless experimentation framework.

CCS CONCEPTS
• Security and privacy → Systems security; Software security engineering; Software reverse engineering; • Networks → Application layer protocols.

KEYWORDS
Bluetooth, macOS

ACM Reference Format:

1 INTRODUCTION
The macOS Bluetooth stack is an interesting research target, since all iMacs and MacBooks exclusively use Broadcom Bluetooth chips. These chips allow unsigned temporary firmware patches via InternalBlue [6]. Integrating macOS into InternalBlue enables full access to the Bluetooth chips in hundreds of millions of devices.

Officially, Bluetooth access on macOS is supported by the CoreBluetooth and IOBluetooth frameworks. However, these frameworks are very restricted. Firmware modification requires access to the Host Controller Interface (HCI), and sending arbitrary data over-the-air requires Asynchronous Connection-Less (ACL) injection. We reverse-engineer the macOS Bluetooth stack to understand HCI and ACL in Section 2. Based on the reverse-engineering results, we develop custom hooks that could also be extended for other applications in Section 3. We conclude the results of the macOS InternalBlue integration in Section 4.

2 BLUETOOTH STACK OVERVIEW
An overview of the macOS Bluetooth stack is shown in Figure 1. User-space applications do not interact directly with the chip, macOS restricts communication to the IOBluetoothFamily.kext driver, running in kernel-space. The official macOS Bluetooth API does not allow sending HCI, ACL, or Synchronous Connection-Oriented (SCO) packets. However, CoreBluetooth and selected publicly documented classes and functions of IOBluetooth offer application developers high-level access to a few very basic functions, i.e., retrieving the Bluetooth address [2, 3]. Playing music via Bluetooth headphones is abstracted further, as the application developer only needs to be aware of music playback but not the specific output method. Thus, audio functions can be accessed via AVAudioPlayer, and then, macOS decides if the music is sent to the internal speakers or an external Bluetooth peripheral depending on the audio settings selected by the user. In case Bluetooth headphones are connected and selected, the music is forwarded to BluetoothAudiod. Since it is a separate daemon, it forwards the audio again to bluetoothd via Cross-Process Communication (XPC).

Nonetheless, the various Bluetooth frameworks need to access functions within IOBluetooth, specifically BluetoothHCISendRawCommand for sending HCI commands to the Bluetooth chip and BluetoothHCISendRawACLData for transmitting ACL data. CoreBluetooth only accesses these functions indirectly via bluetoothd, which in turn accesses IOBluetooth.

Note that the function to send HCI commands was reverse-engineered and documented before [1]. However, the existing project only supported selected commands documented in the Bluetooth

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

WiSec ’20, July 8–10, 2020, Linz (Virtual Event), Austria
© 2020 Copyright held by the owner/author(s).
https://doi.org/10.1145/3395351.3401697

Figure 1: Apple’s macOS Bluetooth stack.
We used various reverse-engineering tools and debugging methods to analyze the macOS Bluetooth stack. Initially, we analyzed macOS binaries using Hopper v4 and Ghidra. Most of these binaries were not stripped and still contained most function names, enabling full-text searches for ‘ACL’ and ‘HCI’. As bluetoothd excessively calls the private IOBluetooth framework, this provided us with many insights.

When accessing functions like BluetoothHCISendRawCommand within a project, they need to be declared in an Objective-C header file and bluetoothd has to be imported. By importing the framework, the binary is linked against it—which also includes undocumented methods. However, to call functions within the IOBluetooth framework, not only their names but also their precise arguments and types are required. There are two methods to reverse-engineer these. The most commonly known is runtime analysis with a debugger like lldb. However, we chose another option. Apple provides a Bluetooth PacketLogger in their Additional Tools for Xcode.

By trying different data types and values for the arguments of BluetoothHCISendRawCommand and BluetoothHCISendRawACL, Data and simultaneously checking the logs in PacketLogger, we were able to determine the purposes and data types of each variable and reconstructed the function signatures. As shown in Listing 1, both functions have parameters for a request identifier, the data to be transmitted, and the total command size in bytes. ACL connections are always end-to-end with another device, which is identified by a handle to support multiple ACL connections in parallel. Thus, the ACL function requires an additional handle parameter.

When guessing the correct function signatures, PacketLogger provides immediate feedback as shown in Figure 2. For example, we initially swapped the handle and request identifier. Thus, PacketLogger complains that there is no connection with the handle 0x0172. After swapping these parameters, ACL data can be transmitted successfully.

### 4 CONCLUSION

We tested the resulting macOS InternalBlue port on a high variety of devices, including a recent MacBook Pro 16” Late 2019 on Catalina, going back to an iMac Late 2009 on High Sierra. Thus, even though the IOBluetooth framework is undocumented, HCI and ACL access stay the same across various macOS versions. Moreover, using IOBluetooth operates independently from the underlying transport mode, which can be USB, UART, or PCIe. With the approach described in this demo, InternalBlue works on all these chip variants. This enables Bluetooth security research on various devices.

---

**Listing 1: Reverse-engineered function names.**

<table>
<thead>
<tr>
<th>Handle</th>
<th>Function Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0172</td>
<td>BluetoothHCISendRawACL</td>
</tr>
<tr>
<td>0x0172</td>
<td>BluetoothHCISendRawCommand</td>
</tr>
</tbody>
</table>

**Figure 2: ACL method calls captured with PacketLogger.**

(a) ACL with wrong function parameters.

(b) Successful ACL transmission.

specification and had no external interface, while InternalBlue support requires vendor-specific commands. Moreover, we are the first to reverse-engineer ACL on macOS.

HCI and ACL are slightly different in their functionality. For example, HCI supports configuring the Bluetooth chip. Most HCI commands are not connection-related. In contrast, ACL is used for data transmission within an active connection and, thus, always requires a connection handle. However, communication with the Bluetooth chip’s interface is very similar for both of them. Thus, both use the more generic private BluetoothHCIDispatchUserClientRoutine, which passes them to the IOKit user-space framework [4]. The corresponding function is called IConnectCallStructMethod and finally forwards the Bluetooth packet to the IOBluetoothFamily kernel-space driver using a Mach port. This driver supports various means of transportation to the chip: USB, UART, and PCIe.

The HCI and ACL methods within IOBluetooth are not callable by external binaries because they are not declared in the IOBlue tooth headers. By declaring them in a header file of an Objective-C project and importing the framework, they become callable. While we chose the methods within IOBluetooth to support InternalBlue on macOS, it would also be possible to instead hook into and import IOKit and use the IConnectCallStructMethod, which is even deeper in the stack. With this declaration, any user on macOS is able to execute a binary that calls these functions—no privileged access is required to modify the firmware on the Bluetooth chip.

An alternate approach is to communicate with bluetoothd via XPC [7]. However, our approach bypasses bluetoothd and directly communicates with the chip—no capability checks on the calling process are performed.

Overall, bluetoothd has more of an administrative role on macOS. In contrast, bluetoothd on iOS is located much deeper within the stack [5].

### 3 REVERSE-ENGINEERING TECHNIQUES

We used various reverse-engineering tools and debugging methods to analyze the macOS Bluetooth stack.

Initially, we analyzed macOS binaries using Hopper v4 and Ghidra. Most of these binaries were not stripped and still contained most function names, enabling full-text searches for ‘ACL’ and ‘HCI’.

---

```c
def BluetoothHCISendRawACLData( void *commandData, size_t commandSize, uint32_t handle, uint32_t request);
```
DEMO SETUP

Our demonstration will consist of two parts: (1) a minimal working example to hook IOBluetooth functions on macOS, as well as (2) a video recording of the full InternalBlue integration.

The minimal working example requires access to a macOS computer running Mojave or Catalina and Xcode. This example provides the user with a command-line application that demonstrates how to call private IOBluetooth functions. In contrast, while the InternalBlue integration uses the same mechanism, is way more complex and harder to understand. The code of this example is openly available and we provide detailed installation and usage instructions.

Since not everyone has access to a macOS device or the time to compile a project in Xcode, we will also upload videos of the minimal working example and the full InternalBlue integration.

ACKNOWLEDGMENTS

This work has been funded by the German Federal Ministry of Education and Research and the Hessen State Ministry for Higher Education, Research and the Arts within their joint support of the National Research Center for Applied Cybersecurity ATHENE.

REFERENCES