Demo: Analyzing Bluetooth Low Energy Connections on Off-the-Shelf Devices

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

Michael Spörk
Institute of Technical Informatics
Graz University of Technology
michael.spoerk@tugraz.at

Carlo Alberto Boano
Institute of Technical Informatics
Graz University of Technology
cboano@tugraz.at

Kay Römer
Institute of Technical Informatics
Graz University of Technology
roemer@tugraz.at

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

Abstract

In this demo, we measure Bluetooth Low Energy link-layer statistics on portable off-the-shelf devices. These statistics can be used to debug ongoing connections and implement custom channel blacklisting strategies.

1 Introduction

Link-layer metadata of Bluetooth Low Energy (BLE) connections contains important performance and debugging information. In connection-based BLE, a slave device is connected to a master and these two devices periodically exchange link-layer packets. Each of these link-layer packets carries important information, such as the used BLE data channel, signal strength, and link quality, that can be used to debug the BLE connection or monitor its performance. BLE radio chips may use these information to adapt link-layer parameters to increase the overall connection reliability, but off-the-shelf radios hide these link-layer information and do not forward any information to the BLE host.

While some BLE chips, such as the popular *Nordic Semiconductor nRF52* platform, allow access to link-layer information, BLE chips of smartphones do not provide any link-layer insights. Therefore, experimentation with BLE on off-the-shelf smartphones is cumbersome, as all link-layer information is hidden in the smartphone's BLE radio chip.

In this demo, we fill this gap by providing link-layer statistics of ongoing BLE connections on *Broadcom* radio chips that are widely used in off-the-shelf devices, such as the *Samsung Galaxy S* series, *iPhones* and *MacBooks*, older *Google Nexus* smartphones, and the *Raspberry Pi* series. Towards this goal, we use *InternalBlue* to apply firmware binary patches to *Broadcom* chips [3]. With our firmware patches, we are able to monitor multiple link-layer metrics including the used BLE data channel, Received Signal Strength Indicator (RSSI), packet acknowledgment, and clock of every BLE connection event. Measuring lower-layer statistics on those devices enables experiments with realistic Internet of Things (IoT) scenarios, where the app is running on an *Android* or *iOS* smartphone.

We have successfully used this information to improve the blacklisting of bad BLE channels significantly [4]. Additionally, these statistics can be used in other applications that require a root cause analysis of connection quality. Upon acceptance of this demo, the code to get these lower-layer statistics will be available online [1]. During the demo, we will provide an installation with various IoT gadgets and smartphones to show our BLE statistics patches.

This demo paper is structured as follows. Sect. 2 details how we patched the *Broadcom* firmware to extract lower-layer BLE information. In Sect. 3, we explain how to apply a custom channel blacklisting. Sect. 4 lists the hardware and software requirements to reproduce our demo.

2 Bluetooth Firmware Patching

If a Bluetooth chip has performance or security issues, new firmware can be compiled by the vendor and rolled out with the next operating system update. With *InternalBlue*, we can modify the existing firmware of *Broadcom* and *Cypress* chips. When *Cypress* acquired the IoT division of *Broadcom* in 2016 [2], they released various data sheets containing important information about the chips, as well as the development platform *WICED Studio*. This platform contains symbols for global variables, hardware registers, and function names of *Cypress* evaluation boards. Symbols enable us to locate relevant functions despite missing source code and documentation. We can search for similar functions in other firmware—binary code and hardware register accesses within a function stay similar.

Broadcom firmware pulls wireless data in synch with the Bluetooth clock. Depending on the current connection state, different task callbacks are executed. Once a

[1] https://github.com/seemoo-lab/internalblue/tree/master/examples

```
internalblue$ python examples/s8_rxdn.py
[*] Loaded firmware information for BCM4347B0. Installing BLE patches...
[*] ------------------------------------------------------------------------------
[*] LE event     0, map ffffffff8, RSSI -42:           *
[*]              ^---------------------------- ERROR --------------------------------
[*] LE event     1, map ffffffff8, RSSI -44:                    *
[*] LE event     2, map ffffffff8, RSSI -43:                              *
...
[*] LE event  5537, map fffd0fff8, RSSI -44:                            *
```

Listing 1. BLE receive statistics visualized by *InternalBlue*. The channel of the current event is plotted as x-axis offset.

BLE connection is established, the data reception callback is `_connTaskRxDone`. This callback executes internal logic, including RSSI measurements and checking of missed packets with the Sequence Number (SN) and Next Expected Sequence Number (NESN). For this purpose, a global struct stores the current connection state. We patch the `_connTaskRxDone` function to pass this connection struct to the host with a custom Host Controller Interface (HCI) packet. HCI packets are logged by common operating systems and can be interpreted during runtime with *InternalBlue*. Our patch puts a packet into the HCI queue, which is sent to the host by a different thread; thus, only a minimal time delay is introduced in the time-critical `_connTaskRxDone` function.

3 Channel Blacklisting Caveats

Changes to the channel map are not time-critical. A connection's master can send a link-layer control frame to update the channel map, but the earliest possible time slot to apply it will be six events in the future. Thus, custom blacklisting can be implemented on the host and applied to the chip using HCI.

According to the Bluetooth specification [1, p. 1351], a BLE channel map can be set manually with the `HCI_LE_Set_Host_Channel_Classification` command on the master of a connection. In practice, *Broadcom* chips will apply further blacklisting and whitelisting mechanisms:

- The minimum number of whitelisted channels according to the Bluetooth specification is two [1, p. 2785], and a *Broadcom* chip will ignore a channel map with only one whitelisted channel,
- if the chip is a Wi-Fi combo chip and Wi-Fi is connected to a 2.4 GHz Wi-Fi channel, the corresponding 20 MHz within the Bluetooth channel map are blacklisted immediately by a proprietary *Broadcom* coexistence mechanism, and
- the *Broadcom* chip runs additional internal statistics that blacklist high interference channels.

Broadcom Bluetooth chips regularly and autonomously update the used BLE channel map. Listing 1 shows an exemplary mature BLE connection between a *Samsung Galaxy S8* smartphone and a smartwatch. Most likely, the channels of a nearby interfering 2.4 GHz Wi-Fi access point were blacklisted, even though Wi-Fi was disabled on the smartphone.

4 Hardware and Software Requirements

InternalBlue requires running on a *Broadcom* or *Cypress* chip. In principle, any of these chips can be supported. However, each chip has an individual firmware, and patches need to be ported to those. As of December 2019, the underlying operating systems supported by *InternalBlue* are *Android*, *iOS*, *Linux*, and *macOS*. Thus, the number of potentially supported systems and chips is high. Tab. 1 lists device and system combinations for that we ported and tested the BLE patch.

Table 1. BLE link-layer statistic patch support on off-the-shelf devices with *InternalBlue*.

Device	Chip	Operating system
Raspberry Pi 3	BCM43430A1	Raspbian 07/2019
Raspberry Pi 3+/4	BCM4345C0	Raspbian 07/2019
Eval Board	CYW20735B1	Debian testing 07/2019
Nexus 5	BCM4335C0	Android 7.1.2 12/2018
Samsung Galaxy S8	BCM4347B0	Android 9 05-09/2019

The BLE patch itself requires to locate four functions within the firmware binary. Overall, the patch is almost similar for all firmware versions, and only these function locations need to be replaced.

Acknowledgments

This work has been performed within the LEAD project "Dependable Internet of Things in Adverse Environments" funded by Graz University of Technology and in the context of the LOEWE centre emergenCITY. This work has been funded by the DFG as part of SFB 1053 MAKI, and the BMBF and the State of Hesse within ATHENE. This work was also partially funded by DFG within cfaed and the SCOTT project. SCOTT (`http://www.scott-project.eu`) has received funding from the Electronic Component Systems for European Leadership Joint Undertaking under grant agreement No 737422. This joint undertaking receives support from the European Unions Horizon 2020 research and innovation programme and Austria, Spain, Finland, Ireland, Sweden, Germany, Poland, Portugal, Netherlands, Belgium, Norway. SCOTT is also funded by the Austrian Federal Ministry of Transport, Innovation and Technology (BMVIT) under the program "ICT of the Future" between May 2017 and April 2020. More info at `https://iktderzukunft.at/en`.

5 References

[1] Bluetooth SIG. Bluetooth Core Specification v5.1, Jan 2019.

[2] Cypress. Cypress to Acquire Broadcom's Wireless Internet of Things Business, Jun 2016.

[3] D. Mantz, J. Classen, M. Schulz, and M. Hollick. InternalBlue - Bluetooth Binary Patching and Experimentation Framework. In *The 17th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys '19)*, Jun 2019.

[4] M. Spörk, J. Classen, C. A. Boano, M. Hollick, and K. Römer. Improving the Reliability of Bluetooth Low Energy Connections. In *International Conference on Embedded Wireless Systems and Networks (EWSN)*, Feb 2020.