BlueSpec: Development of an LMP state machine and a stateful black-box BR/EDR LMP fuzzer

Master Thesis

Author: Tom Nijholt

Supervisors: Erik Poll
            Frits Vaandrager

August 2020
Abstract

We take a look at how to fuzz Bluetooth Basic Rate and Extended Data Rate (BR/EDR), also known as Bluetooth classic. Bluetooth LE is an alternative low energy communication technology similar to, but not compatible with Bluetooth BR/EDR. We review the parts of the Bluetooth Core Specification detailing Link Manager Protocol packets and find 3 ambiguities to exemplify complexity and ambiguity in the specification. We analyse and experiment with existing Bluetooth fuzzing frameworks or tools SweynTooth, InternalBlue, Frankenstein for finding both Bluetooth LE and Bluetooth BR/EDR implementation flaws. We find that there is a solution for fuzzing Bluetooth LE to be found in the SweynTooth framework, but that there is no satisfactory solution to fuzz the full stack of arbitrary Bluetooth BR/EDR implementations. There are tools and frameworks, namely Bluetooth Stack Smasher and BluePAss, to fuzz the upper host layers of a Bluetooth BR/EDR implementation namely the protocols L2CAP, ATT and SDP, but there are no solutions to fuzzing the lower controller stack protocols such as Baseband communication and the Link Manager Protocol (LMP). InternalBlue is a promising new tool that offers the possibility of sending and observing LMP packets in existing connections. We build on the work of InternalBlue and develop a stateful black-box LMP fuzzer BlueSpec for low level Bluetooth BR/EDR fuzzing with the goal of finding Bluetooth implementation flaws. During the development of BlueSpec we find a potential DOS opportunity with Bluetooth BR/EDR legacy pairing packets and find two devices being vulnerable to the KNOB vulnerability, one of which is a new well reviewed Bluetooth 5 USB dongle. Part of the fuzzer is a LMP state machine that we have developed which supports both slave and master devices.
# Contents

1 Introduction .................................................. 6
   1.1 Motivation .................................................. 6
   1.2 Problem description ........................................ 7
   1.3 Research question ......................................... 7
   1.4 Contributions ............................................. 7
   1.5 Scope ...................................................... 7
   1.6 Outline .................................................... 8

2 Related work .................................................. 9
   2.1 Ubertooth .................................................... 9
   2.2 Bluetooth Stack Smasher ................................... 9
   2.3 BluePAss ..................................................... 10
   2.4 SweynTooth ................................................ 10
   2.5 Internalblue ............................................... 10
   2.6 Frankenstein .............................................. 11

3 Background ..................................................... 12
   3.1 Fuzzing ..................................................... 12

4 Bluetooth BR/EDR ............................................ 14
   4.1 Difference between Bluetooth LE and Bluetooth BR/EDR ... 14
   4.2 General Bluetooth architecture .............................. 15
      4.2.1 Controller stack ....................................... 15
      4.2.2 The Host Controller Interface .......................... 16
      4.2.3 Host stack ............................................. 17
   4.3 Frequency-hopping spread spectrum .......................... 18
   4.4 Connection protocol ....................................... 18
      4.4.1 Inquiry ............................................... 18
      4.4.2 Paging ............................................... 19
      4.4.3 Link Manager Protocol ................................. 20
   4.5 Complexity of the Bluetooth Specification .................. 21
5 Wishlist for a Bluetooth BR/EDR fuzzer

5.1 Focus on Bluetooth BR ........................................ 25
5.2 Host stack fuzzing ........................................... 25
5.3 Controller stack fuzzing ...................................... 25
5.4 Monitoring capability ........................................ 26
5.5 State-based fuzzing .......................................... 26

6 Reviewing existing Bluetooth fuzzing frameworks

6.1 SweynTooth fuzzer .......................................... 28
   6.1.1 Practical experiments .................................... 28
   6.1.2 Conclusion ............................................. 29
6.2 InternalBlue .................................................. 29
   6.2.1 Practical experiments .................................... 31
   6.2.2 Enabling Diagnostic logging ............................. 31
   6.2.3 Vulnerability of two devices to the KNOB attack .... 31
   6.2.4 Conclusion ............................................. 32
6.3 Frankenstein .................................................. 32
   6.3.1 Practical experiments .................................... 33
   6.3.2 Conclusion ............................................. 34
6.4 Selecting a base to build the fuzzer ......................... 34

7 Creating our state-based LMP fuzzer BlueSpec

7.1 Architecture ................................................. 36
   7.1.1 Sending arbitrary LMP packets .......................... 37
   7.1.2 Monitoring ............................................. 38
   7.1.3 Packet verification ...................................... 39
   7.1.4 State machine .......................................... 39
   7.1.5 Packet manipulation .................................... 40
7.2 Future improvements ......................................... 40
7.3 Limitations .................................................. 41

8 Results of fuzzing and development

8.1 Devices vulnerable to the KNOB attack ....................... 44
8.2 State machine ............................................... 44
8.3 Interesting behaviour encountered during fuzzing ............ 45

9 Future work

9.1 Monitorability of Bluetooth devices .......................... 46
9.2 Fingerprinting Bluetooth implementations .................... 46
9.3 Hardware and software platform ................................ 47
9.4 Advanced checking for adherence to specification rules .... 47

10 Conclusions

10.1 Bluetooth explanations and complexity ....................... 49
10.2 Requirements for an ideal Bluetooth BR/EDR fuzzer ......... 50
10.3 Discussion of Bluetooth fuzzing tools ....................... 50
10.3.1 SweynTooth ........................................ 50
10.3.2 InternalBlue ....................................... 51
10.3.3 Frankenstein ....................................... 51
10.4 BlueSpec ............................................. 51

A SweynTooth Experiments ........................................ 53
A.1 Installing the SweynTooth fuzzer ....................... 53
A.2 SweynTooth with SUT a smartphone .................. 54
A.3 SweynTooth with SUT an Ubuntu laptop .............. 54

B Frankenstein Experiments ..................................... 56
B.1 Installing Frankenstein .................................. 56
B.2 Simulating CYW20735 .................................. 57

C InternalBlue Experiments ..................................... 58
C.1 Installing InternalBlue .................................. 59
C.2 KNOB Proof of Concept test ......................... 61
C.2.1 Xiaomi Pocophone F1 smartphone .................. 62
C.2.2 Hommie Bluetooth 5 USB dongle ................. 62
C.3 LMP logging .......................................... 63
C.4 KNOB attack with LMP logging ..................... 63
C.4.1 KNOB attack with LMP logging against Intel Dual Band Wireless-AC 7265 ....................... 64
C.4.2 KNOB attack with LMP logging against Hommie Bluetooth dongle ............................ 65
C.4.3 KNOB attack with LMP logging against Pocophone F1 .......................... 66
C.5 Sending custom LMP messages in existing connections .... 66

D Development of our own state-based LMP fuzzer BlueSpec .... 70
D.1 Experiments with tethering .................................. 70
D.1.1 Experiment setup ................................... 70
D.1.2 Tethering with the X10-TWS ....................... 71
D.1.3 Tethering with the smartphone ..................... 72
D.2 LMP fuzzing of a smartphone ............................. 72
D.2.1 Experiment setup ................................... 72
D.2.2 Response of the DUT and evaluation board to sendLmpPacket with varying LMP packets .......... 73
D.2.3 Finding the reason of the crash ..................... 73
D.2.4 Without tethering ................................... 77
D.2.5 Viability of a DOS with LMP_IN_RAND ............. 78
D.2.6 The specification on handling repeated attempts .......... 78
# Acronyms

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACL</td>
<td>Asynchronous Connection-oriented/less</td>
</tr>
<tr>
<td>AFH</td>
<td>Adaptive Frequency Hopping</td>
</tr>
<tr>
<td>AFL</td>
<td>American Fuzzy Lop</td>
</tr>
<tr>
<td>ATT</td>
<td>Attribute protocol</td>
</tr>
<tr>
<td>AMP</td>
<td>Alternate MAC/PHY</td>
</tr>
<tr>
<td>ASB</td>
<td>Active Slave Broadcast</td>
</tr>
<tr>
<td>BB</td>
<td>Baseband</td>
</tr>
<tr>
<td>BCS</td>
<td>Bluetooth Core Scheduler</td>
</tr>
<tr>
<td>BD ADDR</td>
<td>Bluetooth Device Address</td>
</tr>
<tr>
<td>BLE</td>
<td>Bluetooth Low Energy</td>
</tr>
<tr>
<td>BLESA</td>
<td>BLE Spoofing Attacks</td>
</tr>
<tr>
<td>BR</td>
<td>Basic Rate</td>
</tr>
<tr>
<td>CAC</td>
<td>Channel Access Code</td>
</tr>
<tr>
<td>CSB</td>
<td>Connectionless Slave Broadcast</td>
</tr>
<tr>
<td>DAC</td>
<td>Device Access Code</td>
</tr>
<tr>
<td>DSL</td>
<td>Domain Specific Language</td>
</tr>
<tr>
<td>DUT</td>
<td>Device Under Test</td>
</tr>
<tr>
<td>EDR</td>
<td>Extended Data Rate</td>
</tr>
<tr>
<td>EIR</td>
<td>Extended Inquiry Response</td>
</tr>
<tr>
<td>FHS</td>
<td>Frequency Hop Synchronization</td>
</tr>
<tr>
<td>FHSS</td>
<td>Frequency Hopping Spread Spectrum</td>
</tr>
<tr>
<td>GAP</td>
<td>Generic Access Profile</td>
</tr>
<tr>
<td>HCI</td>
<td>Host Controller Interface</td>
</tr>
<tr>
<td>IAC</td>
<td>Inquiry Access Code</td>
</tr>
<tr>
<td>L2CAP</td>
<td>Logical Link Control and Adaptation Protocol</td>
</tr>
<tr>
<td>LAP</td>
<td>Lower Address Part</td>
</tr>
<tr>
<td>LC</td>
<td>Link Controller</td>
</tr>
<tr>
<td>LE</td>
<td>Low Energy</td>
</tr>
<tr>
<td>LL</td>
<td>Link Layer</td>
</tr>
<tr>
<td>LLID</td>
<td>Logical Link IDentifier</td>
</tr>
<tr>
<td>LM</td>
<td>Link Manager</td>
</tr>
<tr>
<td>LMP</td>
<td>Link Manager Protocol</td>
</tr>
<tr>
<td>MAC</td>
<td>Media/Medium Access Control</td>
</tr>
<tr>
<td>PHY</td>
<td>Physical layer</td>
</tr>
<tr>
<td>PoC</td>
<td>Proof of Concept</td>
</tr>
<tr>
<td>PSO</td>
<td>Particle Swarm Optimisation</td>
</tr>
<tr>
<td>RF</td>
<td>Radio Frequency</td>
</tr>
<tr>
<td>SBF</td>
<td>Stateful Black-box Fuzzing</td>
</tr>
<tr>
<td>SDP</td>
<td>Service Discovery Protocol</td>
</tr>
<tr>
<td>SUT</td>
<td>System Under Test</td>
</tr>
<tr>
<td>VM</td>
<td>Virtual Machine</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

1.1 Motivation

Bluetooth is a wireless technology used for positioning, data transfer, audio streaming and as networking solution. Bluetooth technologies are embedded in many aspects of our lives and our environments. It is used in sensitive applications including, but not limited to, access control to buildings and vehicles, communication, control systems, payment solutions, health monitoring and automation of building systems. Furthermore, every year the total annual Bluetooth device shipments have been increasing amounting to 4.2 billion devices being shipped in 2019 [27].

Although there does not seem to be any reports of attacking Bluetooth in the wild, the wireless nature, the sensitive applications Bluetooth is used for and the increasing amount of devices shipped annually makes Bluetooth likely an attractive exploitation target for attackers. The reason why no reports of Bluetooth vulnerability exploitation in the wild are encountered is unknown. Researchers have shown that there are vulnerabilities to be found in the Bluetooth protocol [2] [1] [9] [26] and there are available proof of concept scripts so it is fairly possible to attack Bluetooth devices in the wild.

One effective solution to find security vulnerabilities in software is through fuzzing. For example AFL [33], one of the most popular fuzzers, has found numerous bugs and security vulnerabilities in a wide variety of software over the years. Nearly every operating systems has their own Bluetooth implementation and besides those there are also the wide variety of implementations that are built for embedded applications. Large parts of a typical Bluetooth implementation are generally closed source. We believe that a fuzzing solution for Bluetooth would go a long way towards securing Bluetooth environments preventing damage caused by active exploitation in the future.
1.2 Problem description

It is known that there are quite a few vulnerabilities found concerning Bluetooth technology, either through the Bluetooth specification being inadequately secure or a Bluetooth stack implementation being insecure. Finding these vulnerabilities before attackers do prevents active exploitation. A possible scalable solution could be an universal Bluetooth fuzzer. However, there are not any fuzzing solutions that can fully fuzz entire Bluetooth implementations available.

1.3 Research question

The main motivation for this research is the research question:

*Can we create a full protocol stack fuzz testing solution for Bluetooth?*

To answer this question we first research a variety of fuzz testing solutions and tools. We review the capabilities of currently available Bluetooth fuzzing solutions and tools and determine how we can improve them.

1.4 Contributions

1. We review a section of the Bluetooth Core Specification v5.2 and document 3 ambiguities, see chapter 4.5.

2. We create a list of requirements that we would like to see in a Bluetooth fuzzer, see chapter 5

3. We compare existing Bluetooth fuzzing solutions and discuss their advantages and disadvantages, see chapter 6

4. We discover several devices being vulnerable to the KNOB attack, see chapter

5. We create a state machine of the LMP protocol, see section 7.1.4

6. We present a preliminary version of stateful black-box LMP fuzzer Blue-Spec, see chapter 7

1.5 Scope

Initially our scope consisted of creating a fuzzing solution for entire Bluetooth implementations but due to limited time and unforeseen difficulties our scope was eventually restricted to the fuzzing of the Link Manager Protocol of the Bluetooth BR/EDR technology.
1.6 Outline

We review related work to fuzzing Bluetooth in chapter 2. We explain what fuzzing is in section 3.1. We explain and review important parts of the Bluetooth specification in chapter 4. We review several experimental methodologies and tools aimed at fuzzing Bluetooth Bluetooth BR/EDR and Bluetooth Bluetooth LE in more detail in chapter 6. Using the knowledge gained from experiments with these methodologies and tools in chapter 2, detailed in chapters A, B and C, we attempt to create a faster, more efficient fuzzing solution for Bluetooth BR/EDR in chapter 7. We then discuss preliminary results of fuzzing with BlueSpec in chapter 8. Future work is discussed in chapter 9 and our conclusions are given in chapter 10. All our experiments can be found in Appendix A, B, C and D.
Chapter 2

Related work

In this chapter we discuss research that has been done related to the fuzzing and monitoring of Bluetooth communication. There are two versions of Bluetooth communication technology namely, Bluetooth LE and Bluetooth BR/EDR. We will dive further into the differences and similarities between these communication technologies in chapter 4. Below we review non-commercial tools that provide features for fuzzing both versions of Bluetooth communication. There is also a commercial Bluetooth sniffer called the Ellisys Bluetooth Explorer, but because of its price point rumoured to be above ten thousand dollars we find that it is less interesting.

2.1 Ubertooth

UberTooth [22] is an open source wireless development platform for the Ubertooth One\(^1\), a Bluetooth dongle. Its main feature is the ability to switch to monitoring mode after which it is able to monitor Bluetooth BR/EDR and Bluetooth LE traffic much like the monitoring mode which is available for certain WiFi devices. It allows observers to view some Bluetooth traffic within range of the Ubertooth One. It is not clear from documentation, but it seems that when using the device for monitoring Bluetooth BR/EDR it does not reliably capture all packets and clock values [2] [19]. Due to this unreliability, it seems to be better suited for capturing Bluetooth LE traffic.

2.2 Bluetooth Stack Smasher

Bluetooth Stack Smasher is a Bluetooth L2CAP layer fuzzer developed by Pierre Betouin in 2006 [5]. It can send 11 different L2CAP packets and one fuzzing mode where L2CAP packets with random data are sent. The last packet sent is

\(^1\)The official site of Great Scott Gadgets, developer of the Ubertooth One: https://greatscottgadgets.com/ubertoothone/
kept in a buffer so that if the last packet sent crashes the SUT the responsible packet may be quickly found. It is fairly limited in features being only able fuzz on one higher layer. Furthermore it has not been updated for a long time. We feel that since it is only very small, undocumented and unsupported we should not spend more time researching this tool.

2.3 BluePAss

BluePAss [16] is a tool that was developed to test DoS vulnerabilities of specific software that was running on PDA’s. In their paper Gianluigi Me presents a preliminary version that is able to assemble packets with random strings for any protocol running on HCI, L2CAP and RFCOMM. It sounds promising but no further information could be found on the BluePAss tool nor did it seem that there was further research performed by the author.

2.4 SweynTooth

SweynTooth is a set of Bluetooth LE vulnerabilities that were found with a fuzzing framework. The fuzzing framework is not named directly, so when referring to this fuzzer in the future we will call it the SweynTooth fuzzer. The SweynTooth fuzzer is a Bluetooth LE only fuzzer. It uses custom firmware for a Bluetooth LE dongle to send link layer packets and therefore is able to fuzz on the lowest level of the Bluetooth LE protocol. The fuzzer features a state machine to guide fuzzing and a validation mechanism that validates responses received to the Bluetooth specification. It furthermore fuzzes by sending both fuzzed individual packets as well as fuzzing the Bluetooth LE state machine of the system under test (SUT) by sending unexpected valid messages in different states. Because SweynTooth is the first Bluetooth fuzzer that is able to fuzz lower layers of multiple devices effectively we more elaborately review and discuss it in section 6.1 even though it is limited to Bluetooth LE. We also perform some experiments with the SweynTooth fuzzer in appendix A.

2.5 Internalblue

Internalblue [15] is a Bluetooth experimentation framework based on custom patched firmware for Broadcom/Cypress chips that makes it possible to monitor and inject Bluetooth packets in connections. By reverse engineering and patching the BCM4339 firmware Mantz et al. identified functions that were hooked into to expose the functionalities to higher layer software to facilitate the monitoring and injecting of custom Bluetooth packets. Internalblue is more elaborately discussed in section 6.2 and some experiments are performed with it in appendix C.
2.6 Frankenstein

Frankenstein [23] provides a virtual environment to fuzz Bluetooth firmware. Frankenstein uses Internalblue to extract the current state of a running Bluetooth firmware. This state can then be loaded into Frankenstein where execution can continue. It currently supports mainly the CYW20735 and the CYW20819 Bluetooth development boards. Frankenstein is more elaborately discussed in section 6.3 and some experiments are performed with it in appendix B.
Chapter 3

Background

In section 3.1 we discuss the concept of fuzzing, the differences between white-, grey- and black-box fuzzing and stateful fuzzing.

3.1 Fuzzing

In this section we explain what fuzzing is and the different kinds of fuzzing. Fuzzing is the process of automatically sending input to software in order to find input that causes undefined behaviour of the software.

There are two main methods of generating input for the software. If the inputs are based on an input model or a seed, then it is known as generational fuzzing. If the inputs are based on known accepted inputs, then it is known as mutation based fuzzing.

The observation of the to be tested software may also further define a fuzzing implementation. Here we differentiate between white-box, grey-box and black-box observation models [20]. A white-box implementation has access to the source code, a grey-box implementation only has access to the binary and a black-box implementation can only interact with the software by providing input and receiving output.

A white-box fuzzer typically has access to the source code and may analyse the program behaviour to increase code coverage. An example of this would be using symbolic execution wherein different branches of a program are automatically discovered. Additionally it is possible to compile with sanitizing. Undefined behaviour may not always crash software but a sanitizer can help the software crash more often when undefined behaviour occurs. A sanitizer injects assertions in compile time that causes software to crash when undefined behaviour occurs making it more obvious that undefined behaviour has occurred [25]. There are different sanitizers for different types of undefined behaviour [31]. Using a sanitizer requires access to the source code or binaries and can thus only be used in a white-box or grey-box fuzzer.

A grey-box fuzzer typically has access to the binary and has the ability
to use some instrumentation to obtain diagnostic information about the execution of a program. The instrumentation reports back to the fuzzer about how the software has crashed and indicates that a certain section of code has been reached thereby aiding in tracking code coverage.

A **black-box fuzzer** does not have any information about the internals of the software. It has little overhead as there is not much to analyse about execution or to mutate the input, but since there is no measure of code coverage a black-box fuzzer often does not have the same amount of coverage than white-box and grey-box fuzzers.

**Stateful black-box fuzzing** (SBF) also uses the state the software is in when fuzzing software. The state model can either be generated from messages or a standard or defined by hand. Using a state machine be useful for software that includes a multitude of states or states that might be hard to reach by simply fuzzing, for example an authorised state. Examples of tools that use SBF are Sulley [29] and BooFuzz [6] in academia, and Peach [21] and beSTORM [3] in corporate. In stateful fuzzing, the state model can also be used as a performance metric much like code coverage. The state model can guide the fuzzer to specific states and a specific part of the code base which the researcher is interested in or which have not been covered as extensively.
Chapter 4

Bluetooth BR/EDR

In this chapter we explain what Bluetooth BR/EDR is, the differences between Bluetooth BR/EDR and Bluetooth LE and we explain some Bluetooth BR/EDR lower layer communication protocols relevant to our research. We limit the chapter to only Bluetooth BR/EDR excluding Bluetooth LE to limit the size of this chapter. In further chapters we will mostly discuss and work with Bluetooth BR/EDR thus we have decided to only detail the workings of Bluetooth BR/EDR. All the information in the rest of this chapter except section 4.5 is derived from the Bluetooth Core Specification v5.2 [10].

In section 4.1 we explain the differences between Bluetooth BR/EDR and Bluetooth LE. In section 4.2 we lay out the entire logical architecture of both Bluetooth BR/EDR and Bluetooth LE. In section 4.3 we discuss the Frequency-Hopping Spread Spectrum technology that makes it difficult to fuzz and sniff Bluetooth communication. Finally, in section 4.4 we explain how Bluetooth devices connect to each other.

4.1 Difference between Bluetooth LE and Bluetooth BR/EDR

Bluetooth is a wireless technology for exchanging data. There are two distinctive forms of Bluetooth, namely, Bluetooth BR/EDR and Bluetooth LE. Bluetooth BR/EDR is the oldest form and it is the general purpose format of Bluetooth. Bluetooth LE is a newer form released in 2010 for usage in devices which do not have access to a lot of power. Bluetooth BR/EDR and Bluetooth LE are not inter-operable, but it is possible to build a system that supports both Bluetooth BR/EDR and Bluetooth LE which share a radio (page 187 of [10]).

Bluetooth devices use a range of 80 MHz in the 2.4 GHz band. Bluetooth BR/EDR uses this for 79 physical communication channels separated by 1 MHz. Bluetooth LE uses this for 37 general purpose physical channels and 3 advertising channels. Bluetooth BR/EDR uses physical channels for nearly all communication. A Bluetooth BR/EDR physical channel is organised as a so called
piconet where there is one master device that supplies a clock and hopping pattern and multiple slave devices that follow the master clock and frequency hopping pattern. We will talk more about frequency hopping in section 4.3. A Bluetooth LE channel may be an advertising channel, a connection channel or a broadcast channel. Advertising provides data exchange without setting up a connection. A connection channel uses a piconet setup with frequency hopping, but as opposed to the Bluetooth BR/EDR piconet there may be multiple master devices in the piconet. In Bluetooth BR/EDR physical channels and links are controlled using the Link Manager Protocol (LMP). Bluetooth LE uses the Link Layer (LL) protocol to control physical channels and links.

The basic architecture for Bluetooth BR/EDR consists of the Radio (RF), the Baseband (BB), and the Link Manager Protocol (LMP). For Bluetooth LE it consists of the Physical layer (PHY) and the Link Layer (LL). We will further explain the Bluetooth BR/EDR architecture in section 4.2.

4.2 General Bluetooth architecture

In this section we explain the logical components of Bluetooth BR/EDR. An overview of this general architecture of logical components loosely coupled to the OSI-model can be seen in Figure 4.1. The two major components any Bluetooth system has are the host stack and the controller stack. The controller stack creates connections, directly communicates with controller stacks of other devices and changes properties and the type of the connections created. We will discuss all of the logical components of the Bluetooth BR/EDR system, consisting of the host stack and the BR/EDR controller. We explain the logical components and their functions as it helps viewing the different logical components as fuzzing targets. We explain the architecture from the bottom up, meaning we first explain the controller stack that handles raw connections/links and actually sends packets then follow with the host stack the protocols of which communicate over the links created by the controller stack.

4.2.1 Controller stack

The controller stack is the part of the logical architecture that creates and handles connections. The controller stack consists of the following parts.

1. The Link Controller is responsible for the encoding and decoding of Bluetooth packets.

2. The Baseband Resource Manager handles access to the physical radio. Its main feature is a scheduler that negotiates access to time slots in the physical radio channel to different controller components that require it.

3. The Device Manager manages local device behaviour like the device local name, stored link keys and other functionality. It also performs discovery of devices and initiating connection of devices through the inquiry and
Figure 4.1: The Bluetooth BR/EDR architecture [10] loosely coupled to the OSI-model

The inquiry and paging procedures. The inquiry and paging procedures are described in more detail in section 4.4.1 and section 4.4.2 respectively.

4. The Link Manager manages the created connection and communicates to other link managers using the Link Manager Protocol (LMP). After the initial connection has been made in the inquiry and paging procedure, properties and capabilities of the connection between devices are negotiated by link managers using the LMP. We discuss the LMP further in section 4.4.3.

5. The Physical Radio transmits and receives packets on the physical channel. These are then send through the Link Controller to the Baseband Resource Manager.

### 4.2.2 The Host Controller Interface

Host Controller Interface (HCI) is the standardised communication between the host stack and the controller stack. There are 4 different types of packets.

1. HCI Command packets
2. HCI Event packets
3. HCI ACL Data packets
4. HCI Synchronous Data packets

The host stack issues commands to the controller stack through the HCI using HCI Command packets. Responses to these HCI Command packets and other
information are sent back from the controller stack to the host stack through HCI Event packets. Data from higher layer applications are interchanged between the host stack and the controller stack through HCI ACL Data packets and HCI Synchronous Data packets. In some cases implementation of HCI in the Bluetooth stack is optional for instance if the controller and host stack are implemented on the same microprocessor.

4.2.3 Host stack

The host stack deals with higher layer data. It composes messages and forwards them through HCI commands and data channels to the controller stack. Below we describe all the logical components of a Bluetooth BR/EDR host stack implementation that are specified by the Bluetooth Core Specification v5.2. Most of these components and their interfaces with other logical components are shown in figure 4.1.

1. The Logical Link Control and Adaptation Protocol (L2CAP) multiplexes, segments and reassembles data between the controller stack and the host stack such that data is ready to pass through HCI to the controller stack or go upwards in the stack into different application layers. It handles a set of channels which will be connected with a connection to a remote host through which higher layer data is routed. These channels are created by the L2CAP channel manager which also receives HCI event packets from the controller layer and sends out HCI command packets to the controller layer. Additionally, the L2CAP layer handles the quality of communication channels.

2. The Service Discovery Protocol (SDP) reveals service capabilities information to other devices and makes it possible to view service capabilities information of other devices.

3. The Attribute protocol (ATT) is used to read and write values of attributes on a device with an ATT server. The ATT protocol can be performed as host or as server. As an ATT server, a device can host a set of attributes and make them visible to ATT clients. As an ATT client, the device can read attributes of ATT servers. A device can both be an ATT server and an ATT client at the same time. In a Bluetooth BR/EDR host implementation ATT is optional.

4. The Generic Access Profile (GAP). This part does not directly communicate with other logical components and therefore it is not pictured, but it represents the base functionality common to all Bluetooth devices such as modes and access procedures used by the transports, protocols and application profiles.
4.3 Frequency-hopping spread spectrum

Frequency-hopping spread spectrum (FHSS) is a method of transmitting radio signals used in both Bluetooth BR/EDR and Bluetooth LE. Data is sent over different frequency channels switching channels according to a hopping sequence that is shared between two devices. Bluetooth BR/EDR and Bluetooth LE both use the ISM frequency band, but Bluetooth BR/EDR uses 79 physical channels of 1 MHz and Bluetooth LE uses 40 channels of 2 MHz. Advantages of FHSS usage in Bluetooth are that it prevents eavesdropping and it helps avoiding interference. Eavesdroppers would need access to the hopping sequence to listen to communication or they would have to listen to the entire bandwidth of Bluetooth. Interference may be caused by other transmitters like WiFi transmitters sending radio signals on the same band but by constantly changing frequency this interference is mitigated. To combat interference even more effectively Bluetooth devices can optionally use adaptive frequency-hopping spread spectrum (AFH) in which the health of channels is communicated between devices and is used to adapt the hopping sequence to avoid noisy channels. This frequency hopping is one of the reasons that research into Bluetooth technologies has been difficult. FHSS makes it difficult to monitor Bluetooth communication as an outsider to the communication. As an insider, monitoring communication may also be difficult as large parts of the communication are processed by the controller and are not visible to the host system. The software that runs on these lower layers is generally not accessible to the public.

4.4 Connection protocol

The connection protocol consists of 3 phases, namely inquiry, paging and LMP procedures. After these phases have been successfully completed a connection is fully established.

These phases are handled by the Link Manager and the Link Controller components as can be seen in Figure 4.2.

Initiating a connection consists of discovering the device using the inquiry procedure and subsequently connecting to the device using the paging procedure. After the connection has been initiated, specifics about the connection communication can then be negotiated through LMP procedures. In the following sections we will explain how the inquiry, paging and LMP procedures work starting with the inquiry procedure in section 4.4.1 followed by the paging procedure in section 4.4.2 and finally the LMP procedures in section 4.4.3.

4.4.1 Inquiry

Discovering a device is done through the inquiry procedure which is handled by the Link Controller (LC). The flow of communication and packet types can be seen in Figure 4.3. The main goal of the inquiry procedure is finding out what the device address is of any device in the area. In the inquiry procedure the
device searching for other devices is known as the master device and repeatedly sends out its own Inquiry Access Code (IAC) over different frequencies. A device that is set to be discoverable is known as the slave device and upon receiving the master IAC it will respond with a Frequency Hop Synchronization (FHS) message containing the slave device address and the clock which sets the timing for packets. The slave device may also send an Extended Inquiry Response (EIR) after the FHS message, if it wishes to do so it sets a EIR bit in the FHS message to indicate such. The EIR may contain a local name and a list of supported services. The EIR is an optional message which can decrease the time to actual application communication by removing the need to retrieve information through a remote name request or the Service Discovery Protocol (SDP). After the slave has sent his messages and the master has received them the inquiry procedure is completed. The master and slave now can reliably communicate on the channel indicated by FHS and the clock.

### 4.4.2 Paging

The paging procedure works similarly to inquiry. The flow of communication and packet types can be seen in Figure 4.3. The initiating device is known as the master device and the device being connected to is known as the slave device. The only thing that is required for starting the paging procedure is the device address of the slave device. The clock, that could have been obtained from the inquiry procedure or previous connection and the page scanning mode will accelerate the paging procedure.
The master device first requests a connection from the slave device by repeatedly sending out a message with the slave Device Access Code (DAC). If the slave device is in a connectable state the slave device will check in intervals for any packets containing its DAC. When the slave device receives a message containing its own DAC it will send a message containing its own DAC acknowledging the request back to the master. The master then sends a FHS packet. Upon reception of the FHS packet by the slave, the slave again acknowledges reception by sending its own DAC back. The slave then switches to the master device channel and clock as received in FHS packet. A connection is then essentially established, the master device starts communication on the channel by sending a POLL packet to which the slave device should respond with any type of packet to indicate that the paging procedure was successful.

If the paging procedure has been successfully completed then the slave device is added to the piconet of the master device and the logical transport that is implicitly created between them is known as an Asynchronous Connection-oriented (ACL) logical transport. A broadcasting channel is also available for all the devices in the piconet, this transport is known as the Active Slave Broadcast (ASB).

4.4.3 Link Manager Protocol

The Link Manager Protocol (LMP) is used to control and create logical links and logical transports and also control physical links. Link Managers (LM) use LMP to communicate with each other over either an ACL-C or ASB-C logical link which are respectively ACL and ASB logical links with a Logical Link ID (LLID) of 0b11 that is stored in the payload header of the packets.

Multiple LMP packets in a group form a transaction and these transactions can change properties of a connection, these transactions are also referenced as...
procedures. For setting up an initial connection the general LMP connection establishment that is shown in Figure 4.4 is followed.

![Diagram of LMP connection establishment](image)

Figure 4.4: The general LMP connection establishment procedure. Sourced from the BlueTooth Core Specification V5.2 page 592 [10]

There is one major dividing transaction in this general establishment, namely the host connection request transaction. Before this dividing transaction the devices exchange LMP version, differences in Bluetooth clock, supported features and names. The completion of the host connection request transaction signifies the intention of both devices to set up a connection with the variables negotiated in the earlier transactions. After the host connection request transaction has been completed successfully, procedures regarding security of the connection may be initiated. These include pairing procedures, authentication procedures and encryption procedures. When either device will not initiate any more security procedures it will send a setup complete packet. When both devices have sent a setup complete packet, higher layer traffic may start flowing.

4.5 Complexity of the Bluetooth Specification

The Bluetooth Specification is notoriously complicated [26]. It has been shown that there are flaws in the specification resulting in vulnerabilities in all of the Bluetooth devices [1] [32] [2]. When researching these vulnerabilities it is often noted that the protocol specification is unclear or does not indicate secure procedures. For example, in the August 2020 BLESA [32] attack Wu et al. argue the importance of updating the Bluetooth specification to prevent advanced spoofing attacks as the flaw is located in the specification. The BLESA attack was found by finding a flaw in the Bluetooth connection protocol with the automatic cryptographic protocol verifier ProVerif\(^1\). Similarly in the BIAS [1] attack

\(^1\)https://prosecco.gforge.inria.fr/personal/bblanche/proverif/
Antonioli et al. explicitly note that the vulnerability is in the specification. In both attacks mentioned above the attack were also initially found by finding a flaw in the specification.

In the KNOB [2] research the authors find that it is possible to convince devices to use a 1 byte cryptographic key for securing connections. This attack was also first found in the specification and then tested for in Bluetooth implementations.

Ben Seri and Gregory Vishnepolsky of the Blueborne attacks also note the complexity of Bluetooth and the Bluetooth specification. They attribute the complexity of Bluetooth to the large number of fragmentation layers that can be found in Bluetooth protocols. Fragmentation refers to the cutting up or adding together of packets into smaller or larger packets. For example, the Link Controller layer receives 4 HCI packets and needs to reassemble these into raw packets such that they can be sent over the air. It is optimal to send as few packets as possible but the packets have a maximum size. The Link Controller thus cuts and adds pieces of HCI packets together such that in the end the data of the 4 HCI packets are now represented in 3 raw packets. A diagram of this process can be seen in Figure 4.5.

In some locations there are references to packets that do not even exist. For example, on page 644 the specification indicates that when Secure Simple Pairing certain confirm values do not match then the initiating Link Manager must send a LMP_NUMERIC_COMPARISON_FAILURE packet. However, in the diagram showing said exchange it is noted that the initiating Link Manager sends a LMP_NUMERIC_COMPARISON_FAILED.

In some LMP transactions it is mentioned that when an unexpected packet is received a LMP_NOT_ACCEPTED or a LMP_NOT_ACCEPTED_EXT should be sent back.
However the documentation is not consistent, some transactions don’t mention the usage of LMP\_NOT\_ACCEPTED(EXT) at all even when there is usage of a LMP\_ACCEPTED packet, which makes believe that it should also be possible to send a not accepted variant. It is possible that sole usage of LMP\_ACCEPTED is standard in some transactions as we have not tested this.

In some transactions it is only noted in the text that it is possible to send a not accepted packet, while in others it is provided in the transaction diagrams. An example of this is the Diffie-Hellman key check failure on p.654-655. The text here says that if either side receives an invalid public key or an incorrect confirmation value then it should send a LMP\_NOT\_ACCEPTED. The diagrams showing the LMP transactions for receiving invalid values however only shows that the responder side sends a LMP\_NOT\_ACCEPTED and not the initiating side.
Chapter 5

Wishlist for a Bluetooth BR/EDR fuzzer

In this chapter we detail the functionalities which we would want a Bluetooth fuzzer to have and why it would be beneficial for a Bluetooth fuzzer to have these functionalities. The ultimate goal of a Bluetooth fuzzer would be to be able to fully control the contents of every type of Bluetooth packet sent, to be able to monitor all Bluetooth communication between two devices and to be made aware when a Device Under Test (DUT) Bluetooth implementation is not secure while only having black-box access to the DUT. The user should be able to perform all tests simply by having a compatible Bluetooth device and by wirelessly connecting to the DUT using Bluetooth. For all functionalities below this also means that the attack vector for the remote device is always from lower layers to higher layers.

Figure 5.1: A mockup BlueSpec implementation. In blue: components we wish to fuzz. In green: Host stack. In red: Controller stack.
5.1 Focus on Bluetooth BR

We know from our research in chapter 2 that the SweynTooth fuzzer is already a good option for fuzzing multiple layers of the Bluetooth LE stack. We believe therefore that our efforts are better spent on improving fuzzing of the Bluetooth BR stack. However, we feel that the design of the SweynTooth fuzzer is quite nice. It is therefore that we research the Bluetooth LE fuzzer some more in chapter 6 for inspiration for building our own Bluetooth BR/EDR fuzzer. In the next few sections we will discuss the various parts of Bluetooth BR/EDR we would like to fuzz.

5.2 Host stack fuzzing

As explained in chapter 4.2.3, the L2CAP layer performs multiplexing, segmentation and reassembly of communication between higher and lower layers and receives all its communication from the HCI layer. The L2CAP layer performs minor error checking protecting against the controller falsely accepting packets that contain errors but passed controller integrity checks. There is no documentation on the actual scope of this error checking. It would be interesting to find out precisely which packets are blocked from reaching protocols, like SDP, ATT and others, that are located above the L2CAP layer. Fuzzing this part of a Bluetooth implementation could not only result in vulnerabilities found in the Bluetooth host stack implementation like Fluoride in Android, BlueZ in Linux and so on, but also in the applications or embedded devices that receive information through the host stack implementations.

Ruge et al. [23] have already shown that when HCI messages resulting from LMP fuzzing are sent to a BlueZ host stack it can cause BlueZ to crash. Albeit that these crashes are not confirmed to also be possible when the HCI messages originate from the Bluetooth controller stack as Ruge et al. performed these attacks in a virtual environment where they emulated the Bluetooth controller stack. BleedingTooth [17] has shown that bugs in a Bluetooth host stack implementation can lead to vulnerabilities such as a zero-click remote code execution.

5.3 Controller stack fuzzing

The controller stack handles setting up connections and provides authentication and encryption for these connections. The Link Controller handles setting up connection through the inquiry and paging procedures as described in sections 4.4.1 and 4.4.2. The Link Manager handles negotiating a connection with a remote device through the LMP. With the LMP authentication and encryption of traffic is handled. Fuzzing inquiry and paging procedures may not necessarily decrypt encrypted communication but we theorise that a inquiry or page packet may cause a DUT to respond in unexpected ways. We feel that a buggy implementation of the Link Controller could lead to the controller stack crashing and disrupting the ability of a Bluetooth BR/EDR device to connect to other
devices. Fuzzing LMP may result in finding a wide variety of bugs. Crashing
of the controller stack may occur, but also more nefarious problems may
be found since the LMP handles security procedures like authentication and
encryption but also negotiates transmitting power, channel hopping and test
controls. Errors in security procedures may lead to assumed safe connections
being unsafe. Errors in transmitting power could theoretically lead to breaking
devices. Errors in channel hopping could lead to forced channel usage. Errors in
test controls could lead to exposure to testing functionality. Of course, these are
all just theoretical problems we could find, but for some examples, in particular
the ones concerning security, there is precedence of them occurring [1] [2]. Even
privileged RCEs are possible [23] [17].

5.4 Monitoring capability

Fuzzing works best when you get feedback on whether a fuzzed input results
in undefined behaviour or bugs. In grey and white box fuzzers this ability is
usually implemented through instrumentation and sanitizers but these methods
require access to resources that black box fuzzers do not have. With a black
box fuzzer some weak monitoring of the DUT can be achieved through moni-
toring the response and timeout of the DUT. This does however miss a lot
of undefined behaviour as it is not certain that undefined behaviour actually
triggers a deviation in response and timeout from normal communication. In
theory, monitoring the response and timeout of the DUT would mainly be able
to detect crashing of the DUT. It would be nice if we could implement as much
monitoring as possible to catch as many as possible bugs. Another potential
issue is that Bluetooth uses FHSS as mentioned in section 4.3 which also leads
to difficulty in monitoring communication. Internalblue however causes this
to no longer be a problem as all messages passed between the device running
Internalblue and another remote device can be monitored without any serious
issues.

5.5 State-based fuzzing

A stateful black box fuzzer as described in section 3.1 could theoretically work
great with a remote Bluetooth BR/EDR fuzzing solution. We believe we could
create a state machine of Bluetooth BR/EDR based on the Bluetooth Core
specification wherein transitions in the state machine correspond to communi-
cation between the DUT and the fuzzing platforms. Any non-valid transitions
from a state initiated by the DUT we could then consider undesired behaviour
per the Bluetooth Specification and these transitions could then be investigated
more thoroughly. Using a state machine with Bluetooth BR/EDR fuzzing could
also provide us with all the benefits of state machine based fuzzing as described
in section 3.1 among which a performance metric.
Chapter 6

Reviewing existing Bluetooth fuzzing frameworks

In this section we review and experiment with the SweynTooth fuzzer, InternalBlue and Frankenstein. These are all modern experimental tools and frameworks that are or can be used in fuzzing Bluetooth. These are the tools from chapter 2 that we believe that we can iterate upon because they are actively maintained, open source, provide broad functionality and have some documentation.

We take a look at the SweynTooth fuzzing framework [9] even though it is a Bluetooth LE fuzzing framework because it is a framework for fuzzing the entire protocol stack of Bluetooth LE and it also uses a state machine to guide the fuzzing. They seem to have created a comprehensive and extensible solution to fuzzing the Bluetooth LE stack. We review the SweynTooth fuzzer in section 6.1.

Internalblue [15] showcases a novel method of injecting and monitoring LMP messages by patching firmware of commercial Bluetooth chips. The possibility of commercially available Bluetooth chips, that are already used in devices to facilitate ordinary Bluetooth connections, being able to monitor and inject LMP traffic seems promising to us and we believe this could increase security research into Bluetooth security into the LMP protocol and Bluetooth communication in general. We review Internalblue in section 6.2.

Frankenstein [23] is a framework that provides a virtual environment to emulate Bluetooth device firmware, removing the need for wireless communication when testing a Bluetooth implementation. The ability to control the entire environment around a Bluetooth implementation makes it possible to easily send arbitrary messages to the implementation and to see the effect of it in the internals of the Bluetooth firmware offers more insight into the workings of the Bluetooth firmware. We review Frankenstein in section 6.3.
6.1 SweynTooth fuzzer

SweynTooth [9] is a set of vulnerabilities found with a novel fuzzing framework that can fuzz SMP, ATT, L2CAP and the LL of Bluetooth LE. It uses custom firmware for a nRF52840 in combination with a state machine that guides fuzzing through several stages and layers of the BLE protocol. An overview of the fuzzing methodology can be seen in figure 6.1. Garbelini et al. of SweynTooth developed their own custom firmware for use with the nRF52840 which makes it possible to view all the link layer data that would be hidden from the user if one were to be using the default firmware. Furthermore, the custom firmware makes it possible to send packets with custom contents from the lowest layer, making the full protocol stack of Bluetooth LE available for fuzzing. The authors have integrated its usage with the python package Scapy [4] such that complicated Bluetooth LE packets can be easily crafted. Garbelini et al. also implemented a particle swarm optimisation to optimise mutations in packets towards finding more anomalies. As it is a black-box fuzzer, monitorability is limited. Errors and crashes are detected based on unresponsiveness and non-compliant responses. SweynTooth does not have any functionality for fuzzing Bluetooth BR/EDR and the fuzzing tool is also not yet publicly available, there is only code available to verify the vulnerabilities that Garbelini et al. found. They however mention in their paper that their fuzzer code can be requested by sending an email. So we have send an email and have gotten the source code of their fuzzer such that we can use it in some experiments.

6.1.1 Practical experiments

A limited number of experiments were performed with the SweynTooth fuzzer to understand and research SweynTooth functionality.

1. Installing SweynTooth, see section A.1

2. SweynTooth with a smartphone, see section A.2
3. SweynTooth with a laptop, see section A.3

![Bluetooth LE stack diagram](image)

**Figure 6.2: A Bluetooth LE stack. In red are parts of the Bluetooth LE stack that the SweynTooth fuzzer targets.**

### 6.1.2 Conclusion

The SweynTooth research provides a great fuzzing tool. Having custom firmware for a Bluetooth LE controller allows the SweynTooth fuzzer to have full control over the communication between your own Bluetooth LE controller and DUTs. SweynTooth allows fuzzing of the SMP, ATT, L2CAP and LL protocols of the Bluetooth LE stack. The state machine incorporation allows guiding the fuzzing to finding anomalous behaviour in certain states and the particle swarm optimisation improves the possibility of finding anomalous behaviour even more. The use of Scapy [4] makes the code quite elegant and easy to work with. All in all it is a very impressive system. The only major downside is that it only supports Bluetooth LE.

### 6.2 InternalBlue

InternalBlue [15] is an experimentation framework that makes it possible to monitor LMP messages and inject arbitrary LMP messages in existing connections on selected hardware. The affected part of the controller stack that handles these messages is shown in figure 6.3.

Through reverse engineering of the Broadcom Bluetooth chip BCM4339 Mantz et al. found a patch function that could change firmware operation. With the patch function they hooked into the function that is called for sending messages to make it send messages with contents received in HCI commands. Additionally, they used the patch function to hook into the function that is handling incoming LMP messages and made it send incoming LMP messages in HCI events to the host OS. An application then receives the HCI events for LMP packet monitoring and sends out HCI commands to perform LMP packet injection. A diagram of how this looks in the controller stack is included in figure 6.4. Since sending LMP messages would normally require a custom full stack Bluetooth implementation this approach may be very interesting for
fuzzing with LMP messages. Just as in SweynTooth it might be difficult to find out whether erroneous behaviour is happening in the SUT.

Hardware required for LMP fuzzing is either a Nexus 5, Xperia Z3, Samsung Galaxy Note 3 or CYW20735 evaluation board as there are only assembly patches for these devices to ignore some LMP validity checks\(^1\).

---

\(^1\)https://github.com/seemoo-lab/internalblue/blob/master/doc/features.md

---

30
6.2.1 Practical experiments

We performed several experiments to get acquainted with the tool, establish a LMP traffic monitoring solution and to find out how injecting messages into existing connections works.

1. Installing Internalblue, see section C.1
2. Injecting LMP: KNOB attack, see section C.2
3. Enabling Diagnostic Logging, see section C.3
4. KNOB attacks with LMP Logging, see section C.4
5. Injecting custom LMP messages in existing connections, see section C.5

6.2.2 Enabling Diagnostic logging

Enabling diagnostic logging allows us to monitor LMP traffic in Wireshark which is very useful for debugging purposes. At first sight we believed that we could simply connect to the board, start up Wireshark and proceed to monitor traffic. However, before we were able to monitor LMP traffic we had to install a Wireshark dissector plugin and had to compile an old patched Linux 4.14.111 kernel. More information about this process can be found in section C.3.

6.2.3 Vulnerability of two devices to the KNOB attack

In experimentation with the Internalblue tool we tried out the proof of concept script provided with Internalblue and used it to test whether some devices were vulnerable to the KNOB attack. We tested the following devices:

1. Xiaomi Pocophone F1 on Android 10
2. Intel Dual Band Wireless-AC 7265 on Windows/Ubuntu
3. Hommie Bluetooth 5 USB dongle on Windows/Ubuntu

Pocophone F1 is a recent(2018) smartphone running the latest version of Android. The Intel Dual Band Wireless-AC 7265 is a Wi-Fi and Bluetooth card that is integrated into our laptop. The Hommie Bluetooth 5 USB dongle was the first result on amazon when we were searching for a Bluetooth 5 USB dongle with many(2193) ratings. Eventually we discovered that the Intel Dual Band Wireless-AC 7265 and the Hommie Bluetooth 5 USB dongle on up to date versions of both Windows and Ubuntu(Linux) were vulnerable to the KNOB attack. We informed the authors of the KNOB attack of the vulnerabilities as they requested. The execution and more detailed test results can be found in section C.2 and C.4.
6.2.4 Conclusion

Internalblue presents a low cost solution that is able to monitor and inject LMP traffic and has the potential to provide even more functionality through further reverse engineering and patching of firmware. For this the authors refer to a C based firmware patching framework provided through Nexmon [24]. Other researchers [1] [2] have already experimented with Internalblue and they have shown that Internalblue is a viable platform to perform attacks from. Disadvantages are that it only works with specific devices which have a Broadcom or Cypress Bluetooth chip. It is not assured that this will work on any future devices, since it is dependent on functionality that the manufacturers have added and the manufacturer may decide to alter the code breaking the functionality. The mechanism for patching firmware is also limited to a certain amount of “patchram slots” of which there are, dependent on device only a few left. Mantz et al. of Internalblue found that in the Nexus 5 smartphone only 1.5 KB of RAM and only 16 of 128 patchram slots are available for adding patches. However, it is possible to remove existing patches from the patchram slots and it is also possible to free up some RAM space by shrinking BLOC buffers. Based on the research and the experiments performed we believe that it would be possible to create a basic LMP fuzzer with Internalblue.

6.3 Frankenstein

Frankenstein is software that provides a virtual environment to fuzz wireless firmwares. Frankenstein uses Internalblue to extract the state of a wireless firmware which then can be re-executed in the Frankenstein virtual environment. Ruge et al[23], the authors of Frankenstein, used Frankenstein to fuzz the Bluetooth BR/EDR controller stack implementation of the firmware of the CYW20735B-01 Cypress development board. They found a part of the controller implementation which they called the Bluetooth Core Scheduler (BCS). Through hooking into the BCS they were able to fuzz BCS tasks, Extended Inquiry Responses (EIR) and Asynchronous Connection-Less messages. Fuzzing the EIR in this manner led to them finding a heap buffer overflow in the controller stack implementation which affected numerous devices [8]. Ruge et al. also fuzzed LMP messages by sending LMP sequences to the virtual Bluetooth chip comparing the fuzzing with or without the virtual Bluetooth chip having access to a BlueZ Bluetooth host stack.

Ruge et al. provide fuzzing capability for fuzzing BCS tasks, EIR, ACL and LMP messages on the emulated Bluetooth chip. The logical components of the Bluetooth BR/EDR stack with which the fuzzing capability of BCS tasks, EIR messages and ACL messages are associated with is somewhat hard to determine. The BCS seemingly is a new concept defined by Ruge et al. By comparing a diagram of a detailed Bluetooth BR/EDR architecture from the Bluetooth Core specifications with a diagram of the BCS made by Ruge et al. we believe that the BCS is most likely located in the Baseband Resource Manager.
6.3.1 Practical experiments

1. Installing Frankenstein, see section B.1

2. Frankenstein simulating the CYW20735, see section B.2

With Frankenstein we simulated the CYW20735 firmware and managed to trigger CVE-2019-11516. We have not yet succeeded in validating CVE-2019-13916. Unsure if there needs to be an actual device to connect to or if this is not required. We do obtain segmentation faults but we believe these to be associated with the emulator as the lr (link registers) that being outputted as
debug information are locations of UART functions. Our overall experience with experimenting with Frankenstein is that it is a great tool if one has full access to the firmware, but as Bluetooth firmware is, to our knowledge, always proprietary this limits its use.

6.3.2 Conclusion

Frankenstein is a great solution to fuzzing Bluetooth devices if the firmware running on said Bluetooth device is available. It allows for greater monitoring of the SUT firmware than both SweynTooth and Internalblue by using instrumentation. The main advantage of fuzzing with Frankenstein is that it is possible to view the state of the remote device or the firmware of the remote device as the device is emulated within the environment. The main disadvantage of fuzzing with Frankenstein is that you need to emulate the Bluetooth device firmware, meaning that you first have to obtain the device firmware and then patch it such that it can be emulated. Ruge et al. also implemented a method for replaying LMP sequences based on patched CYW20735B-01 firmware aimed at target hardware, however information about this is limited. It seems that this method is not suitable for fuzzing remote targets.

6.4 Selecting a base to build the fuzzer

In chapter 5 we have determined the type of fuzzer that we would wish to build, in this chapter we have performed research into SweynTooth, InternalBlue and Frankenstein to determine how they work and how we are going to build our Bluetooth fuzzer. We intend to iterate on existing tools and solutions to build a fuzzer that would improve upon the currently available tools. From above research and chapter 2 we have realised that there are already great tools for fuzzing the full stack of Bluetooth LE, namely the SweynTooth fuzzer. However, there is not a tool similar to the SweynTooth fuzzer but for Bluetooth BR/EDR. There are several tools for fuzzing single parts of the Bluetooth BR/EDR stack but there is not a comprehensive solution that can fuzz all layers of the Bluetooth BR/EDR protocol.

One of the reasons why SweynTooth is able to fuzz so many parts of the Bluetooth LE protocol is because it uses custom build firmware for their nRF52840 Bluetooth LE device that makes it possible to send out whatever packet one would want. Obtaining the same for the Bluetooth BR/EDR protocol would be a great step towards creating a fully featured Bluetooth BR/EDR fuzzer as described in chapter 5. Internalblue uses the reverse engineered Broadcom/-Cypress patchram feature to enable monitoring and injection of LMP packets on select Bluetooth BR/EDR devices. It is possible that further reverse engineering of Broadcom firmware could reveal methods to fully control all the packets the device emits. However, the patching mechanism utilised by the Internalblue team is limited to only changing a small amount of code so it is uncertain whether full control can be achieved. Frankenstein built further on
the functionalities of Internalblue to extract state information to create a virtual environment for Bluetooth BR/EDR firmware to run in. In testing the device in this virtual environment they revealed several flaws in the Bluetooth implementation of the device but did not reveal any further possibility of fully controlling the packets the device emits.

We believe that the best platform to extend or to build our fuzzer on would be the Internalblue tool. The ability to monitor and inject LMP traffic of the InternalBlue tool are most aligned with the goals we set out in chapter 5. If we were to opt for building on top of Frankenstein would mean that we would limit out scalability greatly as we would only be able to fuzz one device. SweynTooth is simply no longer an option as we have chosen to work on Bluetooth BR/EDR instead of Bluetooth LE because SweynTooth already offers many functionalities for fuzzing Bluetooth LE. SweynTooth does however offer ideas for additions and a design for a good Bluetooth BR/EDR fuzzer as large features of SweynTooth, such as the state machine based fuzzing and the particle swarm optimisation, could theoretically also be implemented in a Bluetooth BR/EDR fuzzer likely offering similar improvements as they did in SweynTooth.
Chapter 7

Creating our state-based LMP fuzzer BlueSpec

We found that it is not possible to fully realise the ultimate fuzzer as we described it in chapter 5 within the time allotted for this thesis. Since there are limited solutions for fuzzing controller stack communication between devices, as is shown in chapter 6, we believe our time is best spent on improving fuzzing controller stack communication. In this chapter we discuss the goal and creation of BlueSpec, a stateful black-box fuzzing tool that is used for fuzzing the LMP. We will be building it using the Internalblue framework as based on our research in chapter 6 and more specifically chapter 6.4 as we believe it to be best suited as a base to build the fuzzer on and to provide the best functionality for usage in a fuzzer. We first discuss the architecture of our system in section 7.1 and then discuss how we intend to implement the different parts of the fuzzing tool in the sections following thereafter.

7.1 Architecture

We want to fulfil as many of the goals in chapter 5. To do so we aim to build on the Internalblue experimentation framework with an LMP state machine based on the official Bluetooth BR/EDR specification [10]. In SweynTooth we have seen the usefulness of a state machine and we believe that the Internalblue fuzzing tool would benefit from this functionality as well. With the implementation of a state machine we would be able to guide the fuzzing towards more interesting states and automatically verify whether the responses of the DUT are valid. This makes it possible to more quickly find deviant responses and additionally scale the fuzzing more easily to test multiple Bluetooth BR/EDR implementations of different devices. In figure 7.1 one can see the architecture of our fuzzing solution.

Our architecture is mostly inspired by the SweynTooth fuzzing architecture. A session consists of connecting to the DUT and then exchanging packets until
the connection is lost. A packet is received through Internalblue by using a HCI connection with the CYW27035 evaluation board. It is then processed by our communication controller which verifies whether the packet is a valid packet for the current state machine configuration by using the packet verification function. If the packet is valid the communication controller updates the state machine to a new state. The communication controller then obtains new packets to send by using the updated state machine and the packet manipulation function to generate mutated packets based on valid packets. Additionally, a set of redundant packets is generated by keeping track of all the valid packets sent which the communication controller can also choose to send. The communication controller then sends either a mutated packet or a redundant packet through Internalblue and adds the sent packet to a list which holds all the packets sent in the current session. By keeping a list of the sent packets we would theoretically always be able to reproduce a found bug. The communication controller then waits for a response packet and the cycle continues until the connection is lost. When the connection is lost a new session is started anew.

### 7.1.1 Sending arbitrary LMP packets

We can send LMP packets using the `sendlmp` function of Internalblue. The `sendlmp` function adds a custom packet to a packet queue handled by the
firmware. The firmware checks some aspects of the custom packet and can filter out some abnormal packets before it is sent out, like for instance packets where the size of a parameter is too large. The fuzzlmp function of Internalblue disables these filters to allow for sending arbitrary LMP payloads. To facilitate sending valid packets as well, we have created a list of all valid LMP packets as described in the Bluetooth Core 5.2 specification.

**Continuous connection (or Connection stability)**

We found that Bluetooth connections are only active for as long as is required. If no data or purpose is specified for a connection then the connection will quickly drop. In our first tests we would send several LMP

NAME

RES

packets to the DUT which were, rightly so, ignored by the DUT and caused no delay in the DUT sending a LMP

DETACH

packet and subsequently disconnect the connection. This might pose a problem when fuzzing LMP packets. Consider for example the case where a collection of packets are simply ignored by the DUT. The fuzzer might try extending the collection with more packets in an attempt to elicit a response, but these attempts would be fruitless as the first packets in the collection cause the rest of the collection to be disregarded. The right course of action would be to alter the first packets in the collection such that the connection does not disconnect.

An alternative would be to start a connection with a purpose that requires keeping the connection open. Dennis Mantz et al. [15] mention this in Chapter 6.2.2 of their paper and it is also mentioned in the Internalblue github repository issues [12]. The solution is to use tethering, connect a Bluetooth keyboard/headset, use l2ping\(^1\) or something similar that requires a continuous open connection.

We have attempted to setup tethering connections with a smartphone and Bluetooth earphones in section D.1. We experienced that connecting with some Bluetooth devices causes the evaluation board to crash producing a stack dump. Tethering with our smartphone device was only possible when the connection was initiated from the smartphone. Nevertheless, by tethering to a smartphone our connection remained available throughout fuzzing a large variety of packets and when there was no LMP communication as well proving that it is a viable solution to the device disconnecting too quickly.

7.1.2 Monitoring

LMP packets are sent from the evaluation board to the host through HCI vendor diagnostic packets as detailed in section 6.2. InternalBlue then sends these packets to the protocol analyser Wireshark, where we are able to see the LMP packets and their contents. When working on a Linux system to control the evaluation board, the Linux kernel may be a limiting factor in the amount of monitoring that is possible. Using a recent 5.x Linux kernels we were not able to receive vendor diagnostic HCI communication and therefore could not

\(^1\)https://linux.die.net/man/1/l2ping
view LMP traffic. Antonioli et al. created as part of their BIAS research [1] a Linux 4.14.111 kernel patch that when applied can compile a Linux kernel which supports HCI vendor diagnostic packets making it possible to receive LMP packets using Internalblue. In appendix C.3 we describe the procedure of enabling the monitoring of LMP packets by using the H4 BCM dissector Wireshark plugin and a patched Linux 4.14.111 kernel.

Internalblue provides the ability to filter for specific HCI packets that are received. As mentioned in 6.2 the LMP packets are sent over HCI and then processed by Internalblue. These packets have their own signature and we can filter them out and react on them right away requiring only minor additions to Internalblue to process them as we like.

### 7.1.3 Packet verification

Packet validation currently occurs by sending a packet and then registering a HCI filter that checks whether one of the possible correct responses is returned within the timeout period. If the response is correct or expected then we continue fuzzing. If the response is not correct or a reply was not received within the timeout then we record the collection of packets that we have sent to obtain the incorrect response to a log such that the incident may be further analysed later. The timeout for LMP packets is defined as the “LMP response timeout” in the Bluetooth Core Specification v5.2 and it is defined as less than 30 seconds long. But in our experience with sending LMP packets and receiving their replies, the response is almost always within 2 seconds.

### 7.1.4 State machine

To verify received packets and to create valid packets ourselves to send to a target DUT for fuzzing we have created an opcode based state machine of the LMP protocol as can be seen in Figure 7.2. For a more readable diagram of what transitions look like we have also added Figures 7.3 and 7.4. We have created this state machine based on the Bluetooth Core Specification v5.2 [10], mainly Vol. 2 Part C “Link Manager Protocol Specification”. It can handle all incoming and outgoing LMP packets as of Bluetooth 5.2 and switch states in the state machine accordingly. It contains both slave device states and master device states, where the fuzzing Bluetooth device is respectively a slave or a master in the connection. It is opcode based, transitions occur based on the opcode of the packet received or sent. Meaning that when we receive a packet we only verify whether a LMP packet with an expected opcode was provided and when we send a new packet to perform fuzzing it contains the correct opcode. The payload for packets is currently left empty and not filled with explicitly valid data. Currently the default value for packet payloads is set to an all zero payload, which is actually a valid payload for several LMP packet payloads such as packets that only contain a random number.

The software we used to create our state machine is the python transitions package [30]. We chose this software as it is the same software that Sweyn...
Tooth used for building its state machine. The entire state machine consists of 261 states and 672 unique transitions described in about 1100 lines of code. The state machine is so large because transactions, sets of LMP packets interchanged, often differ based on whether the device initiating the transaction is the master device or the slave device. Some transactions are also only possible to be initiated from either the slave device or the master device. Providing states and transitions for both the master and slave device side makes it possible to fuzz Bluetooth BR/EDR devices acting as both a master or slave device. We also publicly release the code for the state machine [18] so that it may be used for other research.

7.1.5 Packet manipulation

Valid response packets or initiating packets opcodes may be inferred from the current state in the state machine. It is however not yet possible to obtain a completely valid packet. That is, one with data that the DUT should accept without doubt.

We believe that it is possible to implement generating valid data per LMP packet as a feature of the state machine in the future. The Bluetooth Core Specification v5.2 defines fields within each type of LMP packet and what its contents should be to aid in this effort and because we can inject and monitor LMP traffic using Internalblue we can also infer valid data through analysing LMP traffic. At the moment we create a mutated packet by obtaining a valid opcode from the state machine and then fill other fields in the packet with random data.

7.2 Future improvements

Improvements we can currently envision for the BlueSpec fuzzing solution are:

- The usage of Particle Swarm Optimisation [13] (PSO). PSO is a method to optimise a problem and has recently been applied to the problem of finding bugs or undesired behaviour with a fuzzer [7] [14]. It has also been used by the authors of SweynTooth [9] and they report that it can provide optimisation over a fuzzing implementation without PSO [9]. We believe that our fuzzer could benefit from it as well such that our fuzzer can more easily find interesting states.

- The usage of packet redundancy in fuzzing. The idea of packet redundancy is to repeat the sending of a valid packet at a later point in time. In the SweynTooth research the authors found that by utilising redundancy they were able to find more non-compliant responses or unresponsiveness by using this strategy. We are curious whether using packet redundancy will have the same effect on finding non-compliant responses or unresponsiveness in Bluetooth BR/EDR devices.
• The state machine can also be improved. Some transitions that the slave and master side have are similar enough where we can eliminate the duplicate states by keeping track of whether the fuzzer is a master device or a slave device through other means. As mentioned in section 7.1.4 payloads are not yet fully generated.

7.3 Limitations

1. The CYW20735 evaluation board is a somewhat unstable platform to perform the fuzzing from. When we are fuzzing a DUT there is a possibility that the LMP packets we send may crash the evaluation board. We are essentially fuzzing the DUT and the evaluation board at the same time.

2. Since we can only start sending packets after a connection is set up, due to the LMP injection functionality of Internalblue, we cannot fuzz during the establishing of a connection.

3. When injecting packets in connections and receiving a packet from the DUT as a response back it is possible that the evaluation board will respond to these packets by themselves. This is likely to interfere with out fuzzing ability as the evaluation board response may derail the entire fuzzing sequence. A solution could be to patch the evaluation board using the patchram feature, that Internalblue uses to implement monitoring, to not respond to LMP packets after having set up a connection but we are not certain how viable this method is as we have not investigated this.
Figure 7.2: The state machine of the LMP protocol. The left side contains all transitions for a master device and on the right side all the transitions for a slave device. Transitions that are possible from every state are left out for readability.
Figure 7.3: Security subsection of the master device section of the state machine.

Figure 7.4: Some of the simple states in the master device section of the state machine.
Chapter 8

Results of fuzzing and development

In this section we present some results we achieved during the work of this thesis.

8.1 Devices vulnerable to the KNOB attack

In the process of us getting acquainted with Internalblue we have checked the vulnerability of several of several devices to the KNOB [2] vulnerability. We tested 3 devices:

1. Xiaomi Pocophone F1, running Android 10
2. Intel Dual Band Wireless-AC 7265
3. Hommie Bluetooth 5 USB dongle, a popular Bluetooth dongle.

We found that both the Intel Dual Band Wireless-AC 7265 and the Hommie Bluetooth 5 USB dongle were vulnerable to the KNOB attack. For more information see section C.2 to C.4.3.

8.2 State machine

As part of creating a fuzzer for the LMP protocol we have created an extensive LMP state machine. It contains both slave device states and master device states, thus it is possible to fuzz a DUT acting as a slave device or as a master device. The state machine is created using the python transitions [30] package. We also publicly release the state machine so that it may be used for other projects [18].
8.3 Interesting behaviour encountered during fuzzing

During the creation and the testing of BlueSpec we found several interesting interactions. There are limited results as the state machine and fuzzer have until recently still been in active development.

1. Repeated LMP\_NAME\_REQ sent towards a Pocophone F1 causes the name in the corresponding LMP\_NAME\_RES response message to being slowly pushed out. Pocophone F1 → ocophone F1 → cophone F1. This happens because the name offset field in the LMP\_NAME\_REQ packets we send changes after we have sent the LMP packet multiple times even though the LMP packet we request Internalblue remains the same. An additional oddity that we encountered is that at first our empty LMP\_NAME\_REQ packets were responded to with a LMP\_NOT\_ACCEPTED, but after sending the identical message several time we encountered that the response sometimes changed to LMP\_NAME\_RES.

2. Sending repeated LMP\_IN\_RAND messages to a Pocophone F1 running Android 10 causes repeated popups requesting a pairing pin. This packet type relates to the legacy pairing procedure of LMP. It is normal that such a popup would appear as it would look we would be trying to initiate a pairing, but when a smartphone receives repeated pairing requests then we would imagine that the popups would scale down or the device would limit the amount of pop ups more rigorously than we encountered. More information about our research surrounding this can be found in section D.2.3 to D.2.6.
Chapter 9

Future work

9.1 Monitorability of Bluetooth devices

Monitoring LMP traffic with Internalblue is currently only possible with the
4.14.111 kernel as this is the only kernel where modifications are crafted for and
this seems not easily portable to newer linux kernel versions. Developing a kernel
patch for recent versions of the linux kernel would improve usage, compatibility
with other applications and security of the machines researchers work with.
Even better would be to make these patches part of the linux kernel such that
there would be no need to patch your kernel yourself after every kernel update.

Finding memory corruptions with a methodology similar to ours and that
of SweynTooth is difficult as the main vector of detecting any malfunctioning of
a Bluetooth DUT is by detecting a timeout in the connection to the device. A
memory corruption does not necessarily cause a system to crash and therefore
it is likely that many memory corruptions would remain undetected. With the
black box approach to fuzzing we are not aware of a better way to detect memory
corruptions that detect these timeouts however.

If we were to have more access to the Bluetooth devices than is currently
available one could perhaps implement a memory sanitizer or other sanitizer
that could alert the tester when a memory corruption occurs. Ruge et al. found
that the firmware of the CYW20735B-01 is based on the RTOS ThreadX, which
can be compiled with an integrated stack sanitizer, but exploit mitigations for
the heap are not possible. Implementing and gaining access to these sanitizers
would provide a greater level of monitorability during the fuzzing of Bluetooth
devices.

9.2 Fingerprinting Bluetooth implementations

Fingerprinting Bluetooth implementations running on devices could be use-
ful for finding out whether devices are vulnerable to known vulnerabilities.
When Bluetooth devices connect to each other they exchange a supported Blue-
tooth Core version number, Company ID and a Subversion Number using the LMP\_VERSION\_REQ and LMP\_VERSION\_RES. We can read these parameters using Internalblue and this information could certainly be used to identify whether a device is vulnerable to a specific vulnerability. For example, the BLURtooth\textsuperscript{1} vulnerability is limited to devices running Bluetooth Core 4.2 through 5.0 \cite{28}. The supported Bluetooth Core version number would already be enough to verify whether a DUT would be vulnerable to BLURtooth. There has already been some research towards fingerprinting Bluetooth devices \cite{11}, but the technology does not seem to be available for the public and we cannot find any papers on its usability in detecting device vulnerability to exploitation of well known Bluetooth vulnerabilities.

9.3 Hardware and software platform

The CYW20735 evaluation board we use as the hardware platform for fuzzing a DUT is not without flaws. We have experienced crashes if too many messages are attempted to be sent in quick succession. Furthermore, it responds to all messages received, even if those messages are received as a response to a message that we have sent ourselves. This obviously influences control over the fuzzing process as some sequences of messages that we want to fuzz may not be possible because the evaluation board sends its own messages in between. We cannot say whether these problems are a shortfall of the hardware platform or the software platform. It might very well be possible that these problems can be solved by updating Internalblue or updating the evaluation board through Internalblue. In case the hardware platform is the problem perhaps a newer hardware platform would not have these limitations.

9.4 Advanced checking for adherence to specification rules

The specification notes more limitations and rules than we can currently check with our state machine. For example, some transactions are only allowed to be used after Link Managers have exchanged supported features. Optional LMP power control transactions can be performed by using the enhanced power control PDUs, LMP\_POWER\_CONTROL\_REQ and LMP\_POWER\_CONTROL\_RES, or by using the legacy power control PDUs, LMP\_INCR\_POWER\_REQ, LMP\_INCR\_POWER\_RES, LMP\_MAX\_POWER and LMP\_MIN\_POWER. The LMP supported features exchange should define which one of these sets of PDUs to use. One should be able to test whether devices adhere to this by capturing the LMP supported features exchange and then verifying whether the devices use the newest possible power control PDUs and discard power control attempts with the non negotiated power control scheme. It is possible to implement these limitations and rules in the

\textsuperscript{1}antonioli2020blurtooth
state machine that we have built in section 7.1.4, but it also possible to learn these limitations and rules by using an optimisation algorithm such as PSO PSO.
Chapter 10

Conclusions

We have examined the Bluetooth specification mostly concerning ourselves with the Basic Rate sections, more precisely Vol. 2: “BR/EDR Controller” of the Bluetooth Core Specification v5.2. In chapter 4 we explain the general Bluetooth Bluetooth BR/EDR architecture, the Bluetooth host stack, the Bluetooth controller stack and the Host Controller Interface separating the host stack and the controller stack.

10.1 Bluetooth explanations and complexity

We discuss the frequency-hopping spread spectrum scheme that Bluetooth uses as a method to transmit data over radio and its effect on the monitoring of Bluetooth communication in section 4.3. It limits the possibility of monitoring Bluetooth communication using devices not actively partaking in the connection themselves. For a reliable fuzzing solution, the solution must thus be at the same time monitoring communication, reporting all information back and send out fuzzing material at the same time. Moreover, it has not been possible to reliably monitor controller stack or lower layer communication of Bluetooth devices at all. To be able to monitor controller stack communication one would have to have access to the device firmware which is almost always proprietary and unavailable to the public or implement our own Bluetooth controller.

We then explain how the connection between two Bluetooth BR/EDR devices is established in section 4.4. Finally, we discuss the complexity of the Bluetooth specification in section 4.5. Over the years numerous vulnerabilities, like BLESAS [32], BIAS [1] and KNOB, were all traced back to errors in the specification. We determine that the specification is hard to traverse and difficult to read and find 3 errors in the specification that may lead to confusion. Another issue contributing to complexity is the size of the specification, the Bluetooth Core v5.2 specification consists of 3255 pages. Although we limited ourselves to one volume in the specification, Vol.2: BR/EDR Controller, this one volume already consists of about 632 pages.
10.2 Requirements for an ideal Bluetooth BR/EDR fuzzer

We have determined the requirements for an ideal Bluetooth fuzzer in chapter 5. The ultimate goal of a Bluetooth fuzzer would be to be able to fully control the contents of every type of Bluetooth packet sent, to be able to monitor all Bluetooth communication between two devices and to be made aware when a SUT Bluetooth implementation is not secure while only having black-box access to the SUT. We take a look at Bluetooth fuzzing research to determine where we can best contribute our efforts to improve Bluetooth fuzzing capability. The Bluetooth fuzzing research we mainly focused on were SweynTooth [9], Frankenstein [23] and Internalblue [15]. SweynTooth is a Bluetooth LE fuzzing tool and research, while Internalblue and Frankenstein are Bluetooth BR/EDR tools and research.

10.3 Discussion of Bluetooth fuzzing tools

Eventually, we ended up choosing to build a Link Manager Protocol fuzzer based on Internalblue, but the experiments with SweynTooth and Frankenstein did provide us with insights on how to build a Bluetooth fuzzer. We shortly discuss these tools next.

10.3.1 SweynTooth

The SweynTooth fuzzer discussed in section 6.1 is able to fuzz the full protocol stack of Bluetooth LE. It makes use of custom firmware written for a Bluetooth LE transceiver and uses a state machine to guide the fuzzing. The system is however limited to Bluetooth LE. The SweynTooth fuzzer has found a large number of vulnerabilities and has proven its efficacy in doing so. In researching SweynTooth we found several techniques which could improve the efficacy of a Bluetooth fuzzer of our own:

- using a specification based state machine
- using timeouts to determine non-compliant responses
- using a particle swarm optimisation to optimise mutations in packets towards finding more anomalies

We set up the SweynTooth fuzzer and executed a SweynTooth PoC script for CVE-2019-19196 against a laptop with an Intel Dual Band Wireless-AC 7265 in appendix A.3.
10.3.2 Internalblue

InternalBlue, discussed in section 6.2, is able to inject and monitor Link Manager Protocol packets in existing connections. It makes this and other firmware changes possible by patching select Bluetooth chip vendor firmware using newly discovered built-in vendor functionalities. We found that it is indeed capable of performing monitoring and limited injection of LMP packets. One has to work on an old Linux kernel version to actually monitor the LMP traffic. Newer versions of the Linux kernel break this functionality. Using Internalblue we discovered that some devices we tested are vulnerable to the KNOB [2] attack in appendices C.2 to C.4.3. We tested the Xiaomi Pocophone F1, Intel Dual Band Wireless-AC 7265 and a popular Bluetooth dongle, the Hommie Bluetooth 5 USB dongle.

10.3.3 Frankenstein

Frankenstein, discussed in section 6.3, provides a virtual environment to fuzz Bluetooth chip firmware states extracted with InternalBlue. It has mainly been used to fuzz the Bluetooth Core Scheduler and LMP packets against extracted firmware. We experimented a little bit with the software and triggered CVE-2019-11516 to confirm everything was working in appendix B.2.

As mentioned before, we ended up choosing to continue with Internalblue as we quickly realised that Internalblue is the best candidate for fulfilling the fuzzing goals we set out. We limited the experiments with the Frankenstein and SweynTooth tools as we realised that performing numerous experiments would cost too much time. The reasons for choosing Internalblue to work with are:

- SweynTooth already provides an ability to fuzz most of the Bluetooth LE protocols, building a fuzzer for Bluetooth LE protocols that might not even be better than what SweynTooth offers seems meaningless
- Frankenstein is great for fuzzing a single chip’s controller stack, but because you need to extract the firmware in order to be able to fuzz a Bluetooth controller stack it is less scalable than Internalblue
- Internalblue is especially interesting to build a fuzzer based on as Internalblue makes it possible to monitor and inject LMP packets, something that previously involved building your own Bluetooth device as the lower layer Bluetooth firmware and communication is generally inaccessible.

10.4 BlueSpec

Fulfilling all of the goals of our wishlist we created in chapter 5 would take too much time so we limited ourselves to creating a stateful back-box LMP fuzzer. We designed and built our own LMP fuzzing solution and a LMP state machine
in chapter 7. The state machine features both slave device states as well as
master device states such that a fuzzer using it may act as both a slave device
or a master device. The state machine is created with the python transitions
package and it is publicly available at [18]. During the development of the state
machine and in using it we found some odd behaviour that we have documented
in section 8.3.
Appendix A

SweynTooth Experiments

Here we discuss our experiments using SweynTooth. The experiments that we will perform are listed below.

A.1 Installing the SweynTooth fuzzer

A.2 SweynTooth with SUT a smartphone

A.3 SweynTooth with SUT an Ubuntu laptop

Hardware used in the experiments:

We used several pieces of hardware in the experiments which are detailed below.

1. Ubuntu 18.04 desktop, the machine we run our experiments from
2. nRF52840 USB dongle
3. Xiaomi Pocophone F1 smartphone with a Qualcomm 845 chipset with a WCN3990 WiFi/BT chip
4. Ubuntu 18.04 laptop with kernel 4.15.0-121 with an Intel Dual Band Wireless-AC 7265 chip

A.1 Installing the SweynTooth fuzzer

The source code for validating the SweynTooth exploits is hosted on the SweynTooth github repository\(^1\), but this excludes the actual fuzzer used to find the exploits. We possibly wanted to use the SweynTooth fuzzer in some fuzzing

\(^1\)Public github repository of SweynTooth, https://github.com/Matheus-Garbelini/sweyntooth_bluetooth_low_energy_attacks
exercises as well and thus we contacted the authors of SweynTooth [9] and they were able to provide us with the source code of the fuzzer.

Installation of the SweynTooth fuzzing framework was possible through usage of a docker\(^2\) script or a requirements bash script, the latter recommended by the authors. We ran into some issues with the docker script so we went with the requirements bash script.

SweynTooth uses the nRF52840 as the hardware sending out the packets. This specific device is required as SweynTooth uses a custom build firmware for this device to enable sending raw Bluetooth LE LL packets. Installation of the custom nRF52840 firmware was not described but was easily located in the directory structure and more information on its installation was available on the public github repository of SweynTooth.

### A.2 SweynTooth with SUT a smartphone

To confirm that our setup works with the SweynTooth framework we attempt to communicate with the Bluetooth LE stack of a Xiaomi Pocophone F1 smartphone with a Qualcomm Snapdragon 845 chipset with a WCN3990 WiFi/BT chip running Android 10. Our experimental setup is shown in figure A.1. The nRF52840 dongle communicates directly with the controller stack of the WCN3990 Bluetooth chip. The WCN3990 Bluetooth chip then communicates with Fluoride, the Android Bluetooth stack. Fluoride then communicates with whatever application wants to send out BLE data.

With a small adaptation to Telink\_key\_size\_overflow\_py\(^3\) we print the advertising packets received.

We are not able to discover advertising packets originating from the smartphone. The reason for this is that the smartphone does not send any Bluetooth LE advertising packets by default when turning on Bluetooth. It is possible to build an app to send advertising packets on certain smartphones depending on the chipset and whether it supports Android 5.0+.\(^5\)

### A.3 SweynTooth with SUT an Ubuntu laptop

We attempted a similar experiment with an Ubuntu 18.04 laptop. The laptop uses a Intel Dual Band Wireless-AC 7265 chip to communicate over Bluetooth. By using the laptop running Ubuntu 18.04 we have more control over what the chip does. We can simply open a terminal and command the chip through BlueZ

\(^2\)https://www.docker.com/

\(^3\)Telink key size overflow script, https://github.com/Matheus-Garbelini/sweyntooth\_bluetooth_low_energy_attacks/blob/master/Telink\_key\_size\_overflow\_py

\(^4\)Adaptation described in the SweynTooth repository issue “nRF52840 Development Kit not working with the python scripts”, https://github.com/Matheus-Garbelini/sweyntooth\_bluetooth_low_energy_attacks/issues/12#issuecomment-645535723

\(^5\)Android 5.0 Lollipop introduces Bluetooth Peripheral Mode, https://developer.android.com/about/versions/lollipop
to send Bluetooth LE advertising packets out. Using Telink_key_size_overflow.py we can verify that we are able to receive malformed packets through the nRF52840 dongle. We also seem to be able to send malformed packets with the nRF52840 dongle. By executing the provided script we are notified that the Intel Dual Band Wireless-AC 7265 seems to not be vulnerable to the key size overflow CVE-2019-19196.
Appendix B

Frankenstein Experiments

Here we discuss our experiments using Frankenstein. The experiments that we will perform are listed below.

B.1 Installing Frankenstein
B.2 Simulating CYW20735

Hardware used in the experiments:

We used several pieces of hardware in the experiments which are detailed below.

1. Ubuntu 18.04 desktop

B.1 Installing Frankenstein

The installation of Frankenstein is fairly simple.

Listing B.1: Code to install Frankenstein

```
sudo apt install qemu-user gcc-arm-none-eabi gcc-multilib
pip3 install django pyelftools==0.24
python3 manage.py runserver
```

Frankenstein currently supports the CYW20735B1, CYW20819A1 and BCM4375B1 firmware for execution in qemu-arm. To start executing the CYW20735B1 we executed:

Listing B.2: Code to install Frankenstein

```
make -C projects/CYW20735B1
gnu-arm projects/CYW20735B1/gen/execute.exe
```
B.2 Simulating CYW20735

In this experiment we simulate the CYW20735 and attempt to trigger the Extended Inquiry Response Exploit (CVE-2019-11516)\(^1\)

To trigger CVE-2019-11516 we simply run:

```bash
hcitool -i hci1 scan
```

After executing the above command we quickly trigger CVE-2019-11516 as confirmed by the segmentation fault. The full error log of qemu is shown below.

```
Context switch idle -> lm
lr=0x02d12f
lm_handleInqFHS (0x40) lr=0x02cc53
lc_handleInqResult (0x21fb1c) lr=0x041d91
inqfilter_isBdAddrRegistered (0x21fb24) lr=0x0bf04581
inqfilter_isBdAddrRegistered (0x21fb24, 0x0);
:
lr=0x041dc3 inqfilter_registerBdAddr (0x21fb24) lr=0x0bf045e9
inqfilter_registerBdAddr (0x21fb24, 0x0);
Heap Corruption Detected
Posthook
pool = 0x20d368
pool->block_start = 0x221a80
pool->capacity = 0x10
pool->size = 0x0180
*free_chunk = 0x135b7f7f
7f7f5b13 | 02dea5bc0485847bf2760
--- <cut some lines for brevity> ---
qemu: uncaught target signal 11 (Segmentation fault) – core dumped
```

Appendix C

InternalBlue Experiments

Here we discuss our experiments using InternalBlue. During all experiments we have an instance of btmon running to monitor additional communication not captured by the Wireshark instance. The experiments that we will perform are listed below.

C.1 Installing InternalBlue
C.2 KNOB Proof of Concept test
C.3 Diagnostic Logging
C.4 KNOB attacks with LMP logging
C.5 Sending custom LMP messages in existing connections

Hardware used in the experiments:
We used several pieces of hardware in these experiments which are detailed below.

1. Ubuntu 18.04 desktop, the machine we run our experiments from
2. CYW920735Q60EVB-01 development board
3. Xiaomi Pocophone F1 smartphone with a Qualcomm 845 chipset with a WCN3990 WiFi/Bluetooth chip
4. Hommie Bluetooth dongle\(^1\) with a Realtek 8761BUV K1N38H9 GK26. It was the cheapest Bluetooth 5.0 dongle that we could find at the time of writing.

\(^1\)https://www.amazon.nl/Bluetooth-Hommie-Buletooth-Ontvanger-4-0BLE-Technologie/dp/B088KDKEZR
5. Ubuntu 18.04 laptop with kernel 4.15.0-121 with an Intel Dual Band Wireless-AC 7265 chip

6. Windows 10 laptop with an Intel Dual Band Wireless-AC 7265 chip

**Experimental setup**

The experiment setups are shown in figure C.1.

### C.1 Installing InternalBlue

The InternalBlue installation instructions are available at \(^2\) [https://github.com/seemoo-lab/internalblue/blob/master/doc/setup.md](https://github.com/seemoo-lab/internalblue/blob/master/doc/setup.md)

The exact commands we followed to obtain a full development install including assembly and disassembly are:

```bash
  git clone https://github.com/seemoo-lab/internalblue
  cd internalblue
  virtualenv -p python3 venv
  source venv/bin/activate
  pip install --editable .[binutils]
```

The documentation describes that in Internalblue some commands like establishing connections requires a privileged user start Internalblue start the

\(^2\) [https://github.com/seemoo-lab/internalblue/blob/master/doc/setup.md](https://github.com/seemoo-lab/internalblue/blob/master/doc/setup.md)
software. Upon trying to start the software as root we found that this caused an error in the Pyperclip package that Internalblue uses. Pyperclip was seemingly not able to retrieve the “XDG_SESSION_TYPE” environment variable and raises an error causing us being unable to launch Internalblue as root. We circumvented this error by setting the environment variable to the right value right before the check and it solved the problem in our case.

For reference we added:

```python
os.environ["XDG_SESSION_TYPE"] = "x11"
```
to line 567 of pyperclip/__init__.py

We also need to execute several commands to use the development board as a normal HCI device\(^3\). The development board is assigned /dev/ttyUSB0 in our system so our commands were:

```
Listing C.1: Connecting HCI device
stty -F /dev/ttyUSB0 3000000
btattach -B /dev/ttyUSB0
```

When starting up Internalblue after these commands we can see that the tool has recognised our development board and has loaded firmware adjustments onto the board to have the board interact with Internalblue.

The documentation recommends the following set of actions to confirm that everything is in working order:

```
wireshark start
count ff:ff:13:37:ab:cd
sendlmp 01 -d 02
```

This starts a Wireshark instance showing us various HCI_CMD and HCI_EVENT messages. We see that the host stack is requesting to create a connection and the controller stack returning connect complete. The sendlmp command fails as of yet, but we assume that this is because there is no device with the specified MAC address available and the documentation noted that LMP messages could only be injected in existing connections. We now enable the Bluetooth on our Xiaomi Pocophone F1 smartphone to verify that it does work when a connection is made. This is setup A in figure C.1. We close Wireshark as to have a clean capture for the following experiment. We execute all the commands in quick succession as the documentation stated that the timing for the sendlmp command may be quite time sensitive. We execute the following:

```
wireshark start
count <BD ADDR>
sendlmp 01 -d 02
```

This time we got a connect complete followed quite quickly by a disconnect complete. We then send the sendlmp command but receive the same error

\(^3\)https://github.com/seemoo-lab/internalblue/blob/master/doc/linux_bluez.md
message. We believe that we must execute the sendlmp command as soon as the connect complete line is shown for success. We try again with this in mind.

This time we have managed to execute the sendlmp command before receiving disconnect complete. In Wireshark we see the difference between these two interactions in the order and size of several HCI_CMD and HCI_EVENT messages. In the case where we sent the command before the disconnect message we see that there are several CMD and EVENT messages with opcode 0xFC4D and a final set with opcode 0xFC58 before the host sends a disconnect message and the other were we failed to execute sendlmp before the disconnect we can see that these sets of messages are sent after the disconnect command. There are two large opcode 0xFC4D EVENT messages in the case where we executed the sendlmp command before the disconnect where there are none of those in the case where we executed the sendlmp command after the disconnect. We later found out that 0xFC4D indicates a read RAM operation initiated by Internalblue.

We try the same experiment with a Hommie Bluetooth dongle and obtain the same results. This is setup B in figure C.1. We believe that this is the correct response but are not completely sure, we are looking into performing some examples to further confirm that our setup is now functioning as expected.

C.2 KNOB Proof of Concept test

In this experiment we test whether we are able to run the KNOB [2] Proof of Concept (PoC) example included in the Internalblue repository [15] and whether the target devices are vulnerable to the KNOB attack.

\[\text{Figure C.2: The KNOB attack}\]

The KNOB attack is a key downgrade attack. We explain the attack shortly using the good users Alice and Bob and evil user Charlie. The attack is also
pictured in Figure C.2. Alice and Bob have created a Bluetooth pairing between them and initiate a connection. Alice requests to activate encryption on the connection and Alice’s Bluetooth controller starts negotiating an encrypted connection and sends a proposed key entropy to Bob with a key entropy of 16 bytes. Charlie intercepts the proposed key entropy message and changes the proposed key entropy of 1 byte. Bob accepts the key entropy request of 1 byte and sends an accept message to Alice. Charlie intercepts the accept message and sends a proposed key entropy of 1 byte to Alice. Alice accepts the proposed key entropy thinking that Bob only supports one byte of key entropy and sends a accept message to Bob. Charlie intercepts the accept message and drops it. Alice and Bob’s Bluetooth controllers then continue communication with each other to create a key with an entropy of 1 byte. For more information on this attack we refer to the article [2] by Antonioli et al.

The PoC installs a patch in the CYW20735 firmware that alters the key entropy used for initiating a connection. If we then initiate a connection from the CYW20735 to the target device and the target device accepts the connection request the target device is confirmed vulnerable to the KNOB attack.

C.2.1 Xiaomi Pocophone F1 smartphone

The target device for the first test is a Xiaomi Pocophone F1 smartphone with an up to date Android version per the 3rd of March 2021. We execute the following commands to start Internalblue and the PoC example.

```
sudo internalblue run_pyscript examples/eval_cyw20735/KNOB_PoC.py wireshark start connect <BD ADDR>
```

In Wireshark we can see HCI traffic indicating that the connect command is received and executed. After a few seconds a HCI event informs us that the connection has timed out which btmon confirms. This is the behaviour that we would expect in a device that is not vulnerable against the KNOB attack. The target device recognises that the entropy of the connection key is not valid and it ignores the connection request. We believe that the smartphone is not vulnerable to the KNOB attack. As the android version of the device is up to date we believe that the vulnerability to the KNOB attack has been patched or it was never vulnerable.

C.2.2 Hommie Bluetooth 5 USB dongle

The target device for the second test is a Hommie Bluetooth 5 USB dongle. We searched on Amazon for a cheap USB 5 Bluetooth dongle and this was the first dongle that showed up at the time. We start a new instance of the Wireshark monitor and issue a connect command with the BD ADDR of the dongle.

We immediately see a lot more communication in Wireshark which we recognise as the Hommie Bluetooth dongle accepting the connection. It is clear that
the dongle is vulnerable to the attack.

C.3 LMP logging

In the Wireshark monitor we are not able to clearly monitor LMP traffic. Being able to view LMP traffic would be very beneficial as it would for example show us whether our messages are valid and how other devices react to them. We believe that it should be possible to fully view the LMP traffic being sent as we have found a packet capture which does show LMP packets in the Internalblue issues section\textsuperscript{4}. We eventually found a github issue\textsuperscript{5} that helped us identify that our problem was a missing Wireshark plugin and a Linux kernel issue\textsuperscript{6}.

The Wireshark plugin was mentioned in the Internalblue repository as an optional installation if one is using an Android device. We however required it as well in our usage of the CYW20735 for viewing LMP messages.

Antonioli et al. of the KNOB \textsuperscript{2} and BIAS \textsuperscript{1} research developed a kernel patch to allow sending and receiving diagnostic messages but it is for Linux kernel 4.14. We can either install the patched kernel and thereby downgrade our Linux kernel or attempt to create a similar patch for our current Linux kernel version. We took a look at the changes to the kernel\textsuperscript{7} and the portability to a more recent kernel by comparing the 4.14 Linux kernel files that are patched to the same files in the recent 5.4.80 Linux kernel. We found that in some instances the files were changed by a lot and thus we decided that it would be easier for us to simply change to the patched 4.14 Linux kernel. However if we find that it becomes important to work on a more recent Linux kernel, for example because of updates that enable usage of some Bluetooth 5 devices, we would likely be able to port the patch to a more recent Linux kernel.

We downloaded the 4.14.111 Linux kernel, patched it with the patch provided by the BIAS authors, compiled it and installed it. Using the patched 4.14.111 Linux kernel we are now able to view LMP traffic in the Wireshark monitor.

C.4 KNOB attack with LMP logging

In the following section we attempt the KNOB attack against multiple devices to obtain some more experience with the Internalblue tool. The devices we will test are:

1. Intel Dual Band Wireless-AC 7265
2. Hommie Bluetooth 5 USB dongle

\textsuperscript{4}Capture provided by user phatut in github issue, \url{https://github.com/seemoo-lab/internalblue/issues/8}
\textsuperscript{5}\url{https://github.com/seemoo-lab/internalblue/issues/27}
\textsuperscript{6}There is a reference to this problem now in the Internalblue repository at \url{https://github.com/seemoo-lab/internalblue/blob/master/doc/linux_bluez.md#diagnostics}
\textsuperscript{7}The Linux kernel patches available at \url{https://github.com/francozappa/bias/tree/master/linux-4.14.111}
3. Pocophone F1

C.4.1 KNOB attack with LMP logging against Intel Dual Band Wireless-AC 7265

In this experiment we attempt to perform the KNOB attack the same way as in section C.2, but this time the SUT is an Ubuntu 18.04 laptop with kernel 4.15.0-121 with an Intel Dual Band Wireless-AC 7265 chip. We hope to now see the LMP traffic during a successful KNOB attack. The setup is shown as setup C in Figure C.1.

Connecting to a laptop was initially rejected giving the error “Connection Rejected due to Unacceptable BD_ADDR”. After setting the laptop Bluetooth device to be “discoverable” and the development board to be “discovering” we were able to connect to the laptop chip. The attack should be performed by pairing with the target device. The CYW927035 that we use as our own controller is patched to attempt to downgrade the encryption strength in such an attack.

We performed the KNOB attack against the laptop and found that the laptop chip is vulnerable to the KNOB attack. This is demonstrated by a packet exchange that we captured that is shown in Figure C.4 and it is also explicitly reported by the Internalblue tool as shown in Figure C.3.

We found an advisory by Intel on the KNOB vulnerability\(^8\). In the advisory it says "Operating system providers and open source software projects have made mitigations available for this vulnerability." Since we did our tests on Ubuntu with Linux kernel 4.14 we thought that there might be some changes in more recent operating systems. To that end we have also performed the KNOB attack against the Intel Dual Band chip connected to an up to date Windows 10 system (up to date as of February 12, 2021). We can confirm that the Intel Dual Band chip is also vulnerable in this configuration. The packet capture of the packets accepting the weakened connection parameters is shown in Figure C.5.

C.4.2 KNOB attack with LMP logging against Hommie Bluetooth dongle

In this experiment we attempt to perform the KNOB attack against the Hommie Bluetooth dongle connected to an Ubuntu 18.04 laptop. The setup can be seen in Figure C.1. Similarly to our experiment in section C.4.1 we set the Hommie Bluetooth dongle to be discoverable and set the CYW27035 to scanning and then initiate pairing. We are promptly informed by Internalblue that the encryption key entropy for the connection was set to 1 byte like in Figure C.3. We can also see that the downgrade of the encryption key is accepted by the Hommie Bluetooth dongle through Wireshark in Figure C.6.

Figure C.6: Successful KNOB attack against the Hommie Bluetooth dongle. After LMP accepted the devices starts communicating over the higher L2CAP layer.
Figure C.7: Failed KNOB attack against the smartphone. Packet 119 indicates that the connection is rejected due to security reasons. Packet 120 indicates to the host that the connection has been disconnected.

We also performed the attack against the Hommie Bluetooth dongle connected to a Windows host system and this setup was vulnerable as well.

C.4.3 KNOB attack with LMP logging against Pocophone F1

In this experiment we perform the KNOB attack against the Pocophone F1 smartphone. This time fully being able to view the LMP traffic. When attempting the KNOB attack against the smartphone we are not informed by Internalblue that the encryption key for the connection is only one byte. When we further investigate what LMP communication occurs between the CYW920735 and the smartphone we find that the smartphone seemingly accepts the connection with an LMP accepted message, but then the CYW920735 informs the host with a HCI event that the connection is rejected due to security reasons as can be seen in Figure C.7. The authors of the KNOB research describe that similar behaviour in Android devices may indicate that they are no longer vulnerable.

C.5 Sending custom LMP messages in existing connections

In order to build a fuzzing solution for the LMP protocol based on the Internalblue tool we attempt to send custom LMP messages using the fuzzlmp and sendlmp functionality included in Internalblue. From researching Internalblue we are already aware of the limitations that it is only possible to send LMP messages in existing connections. We write a script to review the ability of Internalblue to send custom LMP messages. This script can be found in Listing C.2.

9https://github.com/francozappa/knob/tree/master/poc-internalblue
Listing C.2: Code to send two consecutive custom messages

```python
internalblue = HCICore()
internalblue.interface = internalblue.device_list()[0][1]  # just use the first device

# Setup sockets
if not internalblue.connect():
    internalblue.logger.critical("No connection to target device.")
    exit(-1)

internalblue.logger.info("Installing fuzzLmp assembly patch")
# Install patch to send arbitrary LMP messages
internalblue.fuzzLmp()

# Define the first message
opcode = 0x02
payload = b"ABCDEFGHIJKLMNOP"

# Connect to remote device
internalblue.connectToRemoteDevice(b'\xff\xff\x12\x34\x56\x78')

# Flag to indicate whether a connection is completed
internalblue.connection_completed = False

# HCI callback filter to check for HCI event packet of type 'Connect Complete'
def connectionComplete(record):
    
    Adds a new callback function so that we do not need to call Wireshark.
    
    hcipkt = record[0]
    if not issubclass(hcipkt._class_, hci.HCI_Event):
        return

    # Check if event is a Connect Complete Event
    if hcipkt.event_code == 0x03:
        internalblue.logger.info("Connection created")
        internalblue.connection_completed = True
        return

    # Register the HCI callback filter to the main collection of HCI filters
    internalblue.registerHciCallback(connectionComplete)

# Wait until we have an active connection
while not internalblue.connection_completed:
    continue

# Send the first message
internalblue.sendLmpPacket(opcode, payload, True, 0xB)
internalblue.logger.info("Sent first message")

# Change the payload and send the second message
payload = b"1234567890123456"
internalblue.sendLmpPacket(opcode, payload, True, 0xB)
internalblue.logger.info("Sent second message")

# Shutdown ends the receive and send threads.
internalblue.shutdown()
```

In Wireshark we monitor the timing of these messages to determine whether it would be possible to fuzz the LMP protocol with a sequence of messages. Wireshark shows us that the controller sends the messages not immediately one after the other, but other messages are sent in between them see Figure C.8. More specifically we can see that in packets 84 and 85 the controller sends an Information Request and an Information Response to the SUT. The outgoing message queue maintained by the controller thus does not contain our messages consecutively. However, there are no LMP messages to be found in between our two custom messages, indicating that it might be possible to input custom consecutive LMP messages without interference by the controller sending a LMP message on its own accord in between our custom messages.
Figure C.8: Wireshark showing consecutive custom LMP messages in a packet capture. The dark highlighted sections are the HCI commands issued by InternalBlue requesting the sending of custom LMP messages and the custom LMP messages outputted from the controller sent over the air to the SUT.

The packet capture further indicates that our custom messages first start appearing directly after the LMP\_name\_req and LMP\_name\_res packets, see packets 75 and 76. Also pictured is the controller confirming to the host that the HCI commands for sending the custom LMP messages are completed with HCI event packets 77 and 82.
Appendix D

Development of our own state-based LMP fuzzer BlueSpec

In this chapter we detail the experiments we have performed as part of developing BlueSpec.

D.1 Experiments with tethering

D.2 LMP fuzzing of a smartphone

D.1 Experiments with tethering

In this section we detail our experiments with setting up tethering with the goal of obtaining a stable connection on which we could perform fuzzing.

D.1.1 Experiment setup

In the following experiments we try to set up a tethering connection between the evaluation board and two different tethering capable devices. The first device which we try to tether with are X10-TWS earphones and the second device we try to tether with is a Pocophone F1 smartphone.

Hardware and software used in the experiments:

1. Bluetooth earphones (X10-TWS)

2. Xiaomi Pocophone F1 smartphone with a Qualcomm 845 chipset with a WCN3990 WiFi/Bluetooth chip

3. CYW20735Q60EVB-01 Evaluation Board
We run the experiments from a virtual machine which has the following software versions:

1. Ubuntu 20.04
2. Linux kernel 4.14.111
3. BlueZ version: 5.53 (Most recent version for Ubuntu 20.04)

D.1.2 Tethering with the X10-TWS

We first attempt to connect to the earphones using the “connect” command in Internalblue, but this connection unfortunately does not last. As we have encountered some instability before with Internalblue’s connect command we attempt to connect using the BlueZ “bluetoothctl” utility. We seemingly obtain a stable connection as indicated by bluetoothctl and Internalblue, but after a few seconds we see that Internalblue is receiving a stack dump and then crashes due to a bug. We attempt to setup our configuration to try and connect again, but attempting to disconnect causes the bluetoothctl to error. We attempt to remove the earphones from the list of paired devices but this errors as well. Then we receive a notification that the BlueZ Bluetooth daemon **blueoothd** has segfaulted.

Connecting without Internalblue

Without using Internalblue when connecting to the earphones they seem to crash the BlueZ stack as well. The BlueZ Bluetooth daemon again segfaults. Furthermore, even though the earphones indicate that they are connected to the machine we are not able to play audio from the desktop to the earphones.

Connecting without a virtual machine and Internalblue

When removing the fact that we are operating from a virtual machine from the equation we find that establishing a connection with the earphones is about just as troublesome. The device is used and set as the current audio device, but no sound can be heard from the earphones. This time however we can disconnect the earphones, remove it from the list of paired devices and BlueZ did not crash in the process. However, we are not able to connect to the device again and we cannot turn on scanning for Bluetooth devices or turn off scanning for Bluetooth devices having both actions result in the error “org.bluez.Error.InProgress”. We can only assume that a part of BlueZ has errored out and after a few minutes we receive the confirmation of this presumption as we receive another segmentation fault in the BlueZ **blueoothd** daemon. It seems that it is simply not possible to test these earphones with the evaluation board and Internalblue.
D.1.3 Tethering with the smartphone

We attempt to set up a tethering connection to the smartphone like in section D.1.2 and we observe similar behaviour. Again our evaluation board crashes and produces a stack dump. After disconnecting the power to the device for a day however we were able to set up a pairing connection. We note that it was only possible to set up a tethering connection when initiating the connection from the smartphone. Initiating the connection from the evaluation board seems to produce a non-tethered connection which is disconnected quickly.

The smartphone and the virtual machine indicate that the tethered connection is used for audio. The smartphone only indicates audio and the VM indicate that the smartphone is an audio speaker. Curiously, if we sever the connection in the VM, the smartphone believes for quite a while that there is still a connection. Initiating a transfer of files from the smartphone to the VM over the zombie connection does however not produce any reaction in the VM.

D.2 LMP fuzzing of a smartphone

In this section we experimented with LMP fuzzing of the Xiaomi Pocophone F1.

D.2.1 Experiment setup

1. **DUT**: Pocophone F1

2. **Evaluation Board**: CYW20735

The HCI device is setup as follows according to bluetoothctl:

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>cheeseVM</td>
</tr>
<tr>
<td>Alias</td>
<td>cheeseVM</td>
</tr>
<tr>
<td>Class</td>
<td>0x000c0000</td>
</tr>
<tr>
<td>Powered</td>
<td>yes</td>
</tr>
<tr>
<td>Discoverable</td>
<td>yes</td>
</tr>
<tr>
<td>DiscoverableTimeout</td>
<td>0x00000000</td>
</tr>
<tr>
<td>Pairable</td>
<td>yes</td>
</tr>
<tr>
<td>UUID: Headset AG</td>
<td>(00001112−0000−1000−8000−00805f9b34fb)</td>
</tr>
<tr>
<td>UUID: A/V Remote Control</td>
<td>(0000110e−0000−1000−8000−00805f9b34fb)</td>
</tr>
<tr>
<td>UUID: PaP Information</td>
<td>(00001200−0000−1000−8000−00805f9b34fb)</td>
</tr>
<tr>
<td>UUID: Audio Sink</td>
<td>(0000110b−0000−1000−8000−00805f9b34fb)</td>
</tr>
<tr>
<td>UUID: Headset</td>
<td>(00001108−0000−1000−8000−00805f9b34fb)</td>
</tr>
<tr>
<td>UUID: A/V Remote Control Target</td>
<td>(0000110c−0000−1000−8000−00805f9b34fb)</td>
</tr>
<tr>
<td>UUID: Generic Access Profile</td>
<td>(00001800−0000−1000−8000−00805f9b34fb)</td>
</tr>
<tr>
<td>UUID: Audio Source</td>
<td>(0000110a−0000−1000−8000−00805f9b34fb)</td>
</tr>
<tr>
<td>UUID: Generic Attribute Profile</td>
<td>(00001801−0000−1000−8000−00805f9b34fb)</td>
</tr>
<tr>
<td>Modalias</td>
<td>usb:v1D6Bp0246d0535</td>
</tr>
<tr>
<td>Discovering</td>
<td>no</td>
</tr>
<tr>
<td>Discovering</td>
<td>no</td>
</tr>
</tbody>
</table>
D.2.2 Response of the DUT and evaluation board to sendLmp-Packet with varying LMP packets

We set up a simple experiment where we connect to the DUT and then command the evaluation board to send all different possible LMP packets using the Internalblue tool. We then monitor the response of the evaluation board and the DUT using Wireshark and the Internalblue CLI.

Results

The Internalblue CLI and Wireshark indicate that the custom packets which we have defined are sent out. The DUT responds to most of them, except to the packets that are normally only sent as a response to a request such as LMP_NAME_RES or similar. Wireshark warns that all length assertions about the packet formats as they were defined in the LMP packet dissector were not met, but the custom packets are sent nonetheless. We assume that if we were not using the fuzzlmp patch then the evaluation board would not have sent out our packets. Unfortunately, the evaluation board starts a stackdump event right after the LMP packets LMP_DETACH(opcode 7) and LMP_IN_RAND(opcode 8) have been sent. As soon as the evaluation board starts dumping the stack through the Internalblue interface the reception of packets also cedes and we cannot find out what the response of the DUT to the LMP_DETACH and the LMP_IN_RAND packets is.

We have seen from our experiments with Internalblue tethering in section D.1 that the behaviour of starting a stackdump seems to be related to the evaluation board crashing. We assume that one of the sendLmpMessage commands that we have issued has crashed the evaluation board. The evaluation board did not regain the connection out of itself. Powering the evaluation board on and off was required in order to continue experiments.

D.2.3 Finding the reason of the crash

To investigate the reason for the crash we briefly looked into whether LMP_IN_RAND could have caused the crashing of the evaluation board.

Fuzzing without LMP_IN_RAND

We exclude the LMP_IN_RAND message in our fuzzing and attempt to fuzz with all the messages once more. We find that we can now fuzz more messages, but the evaluation board still crashes after a few more messages. The last two messages before the evaluation board stops with sending out messages are:

1. LMP_SRES, opcode 12
2. LMP_TEMP.RAND, opcode 13
3. LMP_TEMP.KEY, opcode 14
After the fuzzing we cannot simply restart Internalblue, but we have to restart the HCI connection between the evaluation board and the host system as well.

**Repeated transmission of LMP_IN_RAND**

We try fuzzing a tethered connection with the only custom message sent being LMP_IN_RAND. This time the evaluation board does not crash. On the DUT we quickly see a popup with the name of the evaluation board asking us to put in a pin as can be seen in Figure D.1.

![Figure D.1: “Firt popup message”](image)

Which is then quickly replaced by another pop-up indicating: In the Wireshark monitor we do not see any difference to an ordinary connection initiated
Figure D.2: “ Couldn’t pair with machine-name because of an incorrect PIN or passkey”
through Internalblue, but the one custom packet we are sending. After attempting this a few more times we find that the second pop-up is replaced by a fading version indicating the same message as before. Additionally this causes the phone to vibrate on the DUT.

**Usage of LMP\_IN\_RAND**

According to page 622-624 of the Bluetooth Core Specification 5.2 [10], LMP\_IN\_RAND is the first packet that is sent to initiate a legacy pairing. The specification indicates that LMP\_IN\_RAND contents should be a random number. We are fuzzing with empty payloads and thus the random number is set to all zero.

The LMP\_IN\_RAND is used for calculating a temporary link key, $K_{\text{init}}$, that is
used for initialisation. It is derived from the BD ADDR, a PIN, the length of the pin and \texttt{LMP\_IN\_RAND} and a function \(E_{22}\).

A \(PIN'\) is derived from the PIN value and the BD ADDR.

\[
PIN' = \begin{cases} 
    PIN[0...L - 1] \cup BD\_ADDR[0...\text{min}5, 15-L] & L < 16 \\
    PIN[0...L - 1] & L = 16 
\end{cases} \quad (D.1)
\]

This eventually evaluates to \(K_{\text{init}} = A'_r(X, Y)\) where \(X, Y\) are defined as:

\[
X = \bigcup_{i=0}^{15} PIN'[i \pmod{L}'] \\
Y = \text{IN\_RAND}[0...14] \cup (\text{IN\_RAND}[15] \oplus L') \\ (D.2)
\]

and \(A'_r(X, Y)\) is a hash algorithm that is a slightly modified version of the \texttt{SAFER+} algorithm. \(X\) is used as the random input and \(Y\) is used as the key for the hash function.

The resulting \(K_{\text{init}}\) is then used to hide random values that are used to create the combination key, which is the same as the link key in the pairing procedure and is used for authentication.

Seemingly fine because it is all based on the PIN. As long as the PIN is sufficiently variable. However, a few days ago a vulnerability in this pairing procedure was discovered\(^1\). Details on the vulnerability are not yet publicly available but the authors held a presentation at WOOT21\(^2\) and a blog can be found that references the CVEs disclosed\(^3\)

### D.2.4 Without tethering

We have setup an alternative experiment where we send opcode 8 messages without having a tethered connection. We initiate a connection and as soon as the connection is completed we send an \texttt{LMP\_IN\_RAND} message. The popup shows on the smartphone screen, then disappears after about 2 seconds and the connection is dropped immediately after.

When we attempt this continuously, reconnecting whenever we are disconnected, we find that we can cause a popup about every 1-2 second(s) with sporadic small intervals of about 5 seconds. If there is no interaction on the smartphone non-fading popups are continuously shown while in the “Pair new device” menu. If in any other place a drawer popup and a faded popup are shown in addition to the device vibrating.

\(^1\)CVE entry on cve.mitre.org, 26May2021
D.2.5 Viability of a DOS with LMP\_IN\_RAND

The change from a regular popup to a faded popup we saw in the previous sections is possibly an adaptation to limit the amount of sequential popups that can take over the entire screen. If this amount were unlimited then the screen could be constantly overtaken by a device trying to connect. However, we feel that it is still an issue that a single remote device can repeatedly trigger these popups on an android device. We suggest that after a while the attacker should perhaps be added to a list of blocked devices, at the very least connection requests should not be able to result in these constant popups.

It is even possible to have popups occur on the DUT when it is not visible to other devices i.e. not in the inquiry mode. When the DUT is already connected to a Bluetooth device it also still possible to cause these popups to occur on the DUT.

However, researching the possibility of attacks using the LMP\_IN\_RAND message remains future work as our main goal is building a fuzzing solution.

Future research could be researching what happens when multiple devices attack a single DUT using this attack. Alternating the evaluation board Bluetooth address between sending LMP\_IN\_RAND messages may also be interesting with regards to the ability of the DUT to switch to faded popups. Testing repeatedly sending LMP\_IN\_RAND against other devices with not such extensive functionality as the smartphone may theoretically disrupt the ability of a legitimate user to connect to the device.

D.2.6 The specification on handling repeated attempts

The specification provides a small section on page 974 [10] on what should occur when repeated attempts at pairing using above method are performed without success. It specifies to implement a waiting interval between authentication attempts initiated by a device claiming the same identity as the failed device. This interval is not defined by the specification but is implementation dependent. Furthermore this reinforces the idea that when a single device sends repeated LMP\_IN\_RAND messages from a spoofed BD ADDR, thus with a different identity, the DUT might not limit pop ups.
Bibliography


