UNIVERSITY OF OULU
FACULTY OF INFORMATION TECHNOLOGY AND ELECTRICAL ENGINEERING

Matias Karhumaa

BLUETOOTH LOW ENERGY LINK LAYER INJECTION

Master's Thesis
Degree Programme in Computer Science and Engineering
June 2021

Karhumaa M. (2021) Bluetooth Low Energy Link Layer Injection. University of Oulu, Degree Programme in Computer Science and Engineering, 36 p.

ABSTRACT

Bluetooth Low Energy is a very widely used short-range wireless technology. During the last few years many high visibility Bluetooth related vulnerabilities have been discovered. A significant amount of them have had an impact on implementations of the lowest protocol layers of Bluetooth in firmware running on separate embedded System on Chip dedicated for wireless communication. Bluetooth LE Link Layer implementations have not yet been under systematic fuzzing by vendors as there has been no mature way to inject fuzzed Link Layer packets over the air to the target device.

The goal of this thesis was to design and implement a solution for Bluetooth Low Energy Link Layer injection to enable fuzzing of Link Layer implementations with Synopsys Defensics, a commercial fuzzing framework. Two different approaches were designed and implemented. Both approaches used vendor-specific HCI commands and events for providing a convenient way to inject arbitrary Bluetooth Low Energy Link Layer packets over the air to target devices and at the same time retaining the normal functionality of the Bluetooth LE dongle. The solution was evaluated against state of the art in this field and the results show that the solution is on par with state of the art in this field.

Keywords: fuzzing, wireless protocol, embedded devices

Karhumaa M. (2021) Bluetooth Low Energy linkkitason injektointi. Oulun yliopisto, Tietotekniikan tutkinto-ohjelma, 36 s.

TIIVISTELMÄ

Bluetooth Low Energy, Bluetoothin vähemmän energiaa kuluttava versio, on erittäin laajasti käytössä oleva lyhyen kantaman langaton tiedonsiirtoteknologia. Viime vuosien aikana julkisuudessa on ollut useita Bluetooth-haavoittuvuuksia. Monet näistä haavoittuvuuksista ovat koskettaneet erityisesti alimpia Bluetooth protokollakerroksia, jotka tyypillisesti toteutetaan langattomalle tiedonsiirrolle erikseen suunnitellulla järjestelmäpiirillä suoritettavassa laiteohjelmistossa. Bluetooth Low Energy-linkkitason toteutuksia ei ole laajamittaisesti ja järjestelmällisesti fuzz-testattu laitevalmistajien toimesta, koska tähän mennessä ei ole ollut olemassa yleistä tapaa injektoida fuzzattuja linkkitason Bluetooth Low Energy-paketteja langattomasti testattavaan laitteeseen.

Tämän työn tavoitteena oli suunnitella ja toteuttaa ratkaisu Bluetooth LE-linkkitason injektioon. Ratkaisu mahdollistaa Bluetooth Low Energy-linkkitason toteuttavien laitteiden fuzz-testauksen käyttäen kaupallista Synopsys Defensics fuzz-testausohjelmistoa. Työssä esitellään kaksi erilaista lähestymistapaa Bluetooth Low Energy-linkkitason injektiomenetelmän toteuttamiseen. Molemmissa tavoissa hyödynnetään valmistajakohtaisia laajennuksia HCI rajapintaan, millä mahdollistetaan vaivaton tapa injektoida Bluetooth LE-linkkitason paketteja langattomasti testattavaan laitteeseen samalla säilyttäen injektioon käytettävän laitteen normaali toimintakyky. Tämän työn puitteissa suunniteltua ja toteutettua ratkaisua vertailtiin alan viimeisimpään kehitykseen ja tulokset osoittavat ratkaisun olevan kilpailukykyinen.

Avainsanat: fuzz-testaus, langattomat protokollat, sulautetut järjestelmät

TABLE OF CONTENTS

FOREWORD

This thesis was completed while working at Synopsys Finland Oy. The idea for this thesis was sparked during fall 2019 when I read a research paper [1] from researchers of Technical University of Darmstadt. With encouraging feedback from my colleagues, I presented the idea to my superiors and was allowed to spend some time researching the topic further. Most of the research and development work was done during the year 2020, mostly working from home due to the COVID-19 pandemic. Writing the thesis and experiments were done during spring 2021.

Big thanks to everyone who supported me during my prolonged university studies and during this thesis work. Special thanks to my colleagues for their continuous support and feedback on technical matters. I would also like to thank Professor Juha Röning and Pekka Oikarainen for their precious feedback during the writing process and Mats Honkamaa for proofreading the thesis.

Finally, I would like to thank my loved ones, my wife, and three daughters for bringing joy and being patient with me.

Oulu, June 7th, 2021

Matias Karhumaa

LIST OF ABBREVIATIONS AND SYMBOLS

ADB	Android Debug Bridge
API	Application Programming Interface
ASLR	Address Space Layout Randomization
ATT	Attribute Protocol
BR/EDR	Basic Rate / Enhanced Data Rate
CRC	Cyclic Redundancy Check
ESP	Executable Space Protection
FHSS	Frequency Hopping Spread Spectrum
GATT	Generic Attribute Profile
HCI	Host Controller Interface
IoT	Internet of Things
IP	Internet Protocol
L2CAP	Logical Link Control Adaptation Protocol
LL	Link Layer
LMP	Link Manager Protocol
MIC	Message Integrity Check
PDU	Protocol Data Unit
PSO	Particle Swarm Optimization
RF	Radio Frequency
SCTP	Stream Control Transmission Protocol
SDK	Software Development Kit
SMP	Security Manager Protocol
SoC	System on Chip
SPI	Serial Peripheral Interface
SUT	System Under Test
TCP	Transmission Control Protocol
UART	Universal Asynchronous Receiver Transmitter
UDP	User Datagram Protocol
USB	Universal Serial Bus

1. INTRODUCTION

Bluetooth is a short-range wireless technology used by billions of devices. During the year 2019, more than 4.2 billion Bluetooth enabled devices were shipped globally and the amount of shipped devices is increasing [2]. Robustness and security of the Bluetooth implementations has become essential as the wide adoption has made Bluetooth attractive also from attackers point of view.

Software testing is a process of trying to find errors, also known as bugs, in software. Every non-trivial software is expected to include implementation or design flaws [3]. The errors in software are not just quality problem but potentially also security vulnerabilities or physical safety issues that could be exploited by bad actors [4 p.75].

Fuzz testing has proven to be a very effective methodology for finding bugs and serious security issues in software. Tools for fuzzing different wireless protocols over the air exist but they are often limited to protocol layers that are accessible through common protocol stack APIs. Fuzzers for protocol layers implemented very close to theradio interface, for example, Bluetooth Low Energy Link Layer, are very rare and currently not commercially available.

Synopsys is a leading provider of commercial protocol fuzz testing tools. They wanted to expand their current wireless protocol fuzzing offering to protocol layers that have not been under fuzzing yet and provided topic of this thesis. Until now there have not been any commonly available hardware and firmware options for injecting Bluetooth Low Energy Link Layer PDUs over the air for fuzzing purposes.

The goal of this thesis is to enable fuzz testing of Bluetooth Low Energy (LE) Link Layer (LL) protocol using commonly available hardware. This is achieved by designing and implementing a novel way to inject arbitrary LL packets over the air. The designed solution extends the standard HCI interface for passing LL packets for transmission and reception. Even though the goal is to enable fuzzing of specific technology, the actual fuzzer implementation is out of the scope of this thesis.

The first part of this thesis gives a brief overview of used technologies and the context where the solution is going to be used. The second part introduces the state of the art of wireless protocol security evaluation tools and describes what kind of vulnerabilities can be expected to be found with the solution proposed in this thesis. The third part proposes two different design and implementation approaches and describes them in detail. The last part of this thesis compares those two approaches and evaluates the experiments against the goals of this thesis. In addition results are discussed in a wider context comparing the results to state of the art in this field.

The purpose of this thesis is to answer to following question:

- Can arbitrary Bluetooth Low Energy Link Layer packets be sent over the air using vendor specific extension to HCI interface?

2. TECHNOLOGY REVIEW

The topic of this thesis applies two different fields of science, wireless Bluetooth technology and software security and quality related fuzzing. To give some context, this chapter provides a brief introduction to both technologies.

2.1. Bluetooth

Bluetooth is a short-range wireless technology used by billions of devices. During the year 2019 more than 4.2 billion Bluetooth enabled devices were shipped globally [2]. There are two variants of Bluetooth which are not directly compatible: Basic Rate / Enhanced Data Rate (BR/EDR) and Low Energy (LE) [5]. This thesis concentrates on Bluetooth LE. The most common use cases for Bluetooth are audio streaming and phone calls. Traditionally, Bluetooth BR/EDR has been used for those use cases but recently Bluetooth 5.2 specification introduced LE Audio which will also drive audio streaming and phone call use cases to Bluetooth LE in the future. According to the Bluetooth Market update, Bluetooth LE remains the fastest growing Bluetooth radio with a 26% compound annual growth rate [2].

2.1.1. Bluetooth LE Architecture

The Bluetooth LE system consists of three main components: host, controller, and radio. Figure 1 shows the high-level architecture of any Bluetooth system.

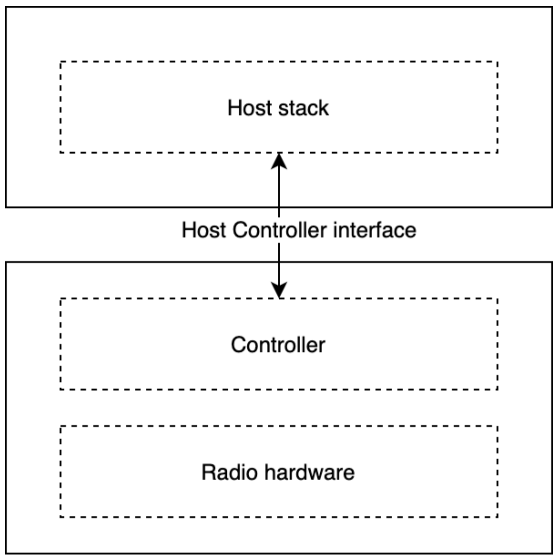

Figure 1. Bluetooth protocol stack main components

Bluetooth LE operates in unlicensed bands at 2.4 to 2.485 GHz using the frequency hopping spread spectrum (FHSS) method for transmitting radio signals. It has been designed for low power use and it provides many protocol parameters for optimising the energy consumption [6]. The frequency band is divided into 40 different 2 MHz wide RF channels. Channels 37, 38, and 39 are dedicated for broadcast, discovery, and connecting. Channels 0-36 are general purpose channels meant mainly for data transmission [5].

Preamble (1-2 octets)	Access-Address (4 octets)	PDU (2-258 octets)	CRC (3 octets)	Constant Tone Extension (optional)

Figure 2. Air interface packet format

Figure 2 shows the Bluetooth LE packet format in the air. The most important field from this thesis' point of view is the PDU field. The PDU field contains the actual protocol data which content depends on the used physical channel. In case of connected communication, PDU contains LL Control or Data PDUs. The goal of this thesis is to be able to send arbitrary LL PDUs over the air. A more detailed description of the LL protocol is in section 2.1.3.

Preamble is one or two octets long fixed sequence of alternating 0 and 1 bits depending on used physical channel. It is used for frequency synchronization and symbol timing estimation on receiver side. Access Address is four octet value used for for indicating the physical channel. In case of data channels, Access Address is used also as unique identifier of connection. Advertising channel packets use fixed Access Address. Three octet long Cyclic Redundancy Check (CRC) is used for error detection. Receiver drops packets where CRC value does not match to one calculated over PDU field received over the air. Constant Tone Extension is optional and only used in direction finding use case.

The Bluetooth LE controller component contains the Link Layer protocol implementation and state machine. The controller takes care of packet transmission and receiving, LL packet scheduling, LL Control Protocol handling, and communication with the host stack through the Host Controller Interface (HCI). The Link Layer protocol is further described in section 2.1.3 and HCI in section 2.1.4.

The host stack implements higher level protocols and profiles and provides APIs for Bluetooth LE application development. The host stack sends HCI commands over USB, UART, or SPI to the controller and receives HCI events from the controller.

Logical Link Control Adaptation Protocol (L2CAP) is the lowest layer of the host stack and most of the protocols and profiles are built on top of it. L2CAP is used for multiplexing multiple channels and fragmentation and reassembly of data. The Bluetooth LE protocol stack is described in section 2.1.5.

There are a few open source host stack implementations available [7, 8, 9, 10]. In addition, many commercial Bluetooth stack options are available.

2.1.2. Bluetooth LE Roles and States

The most relevant parts of Bluetooth LE roles and states in the scope of this thesis are summarized in Figure 3. More detailed information can be found from Bluetooth Core Specification [5].

Non-connected communication called advertising is one-to-many broadcast communication. The device in peripheral role advertises its presence and features by transmitting advertising packets on RF channels 37, 38, and 39. Advertising packets

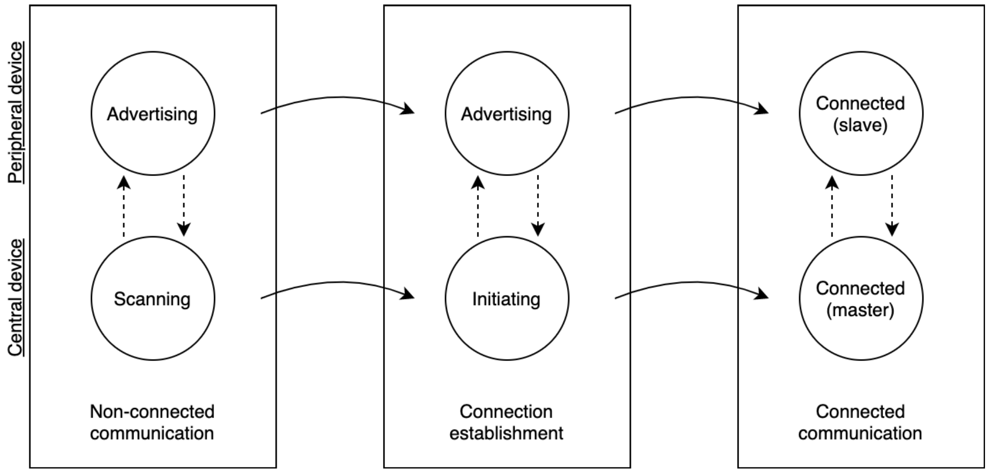

Figure 3. Bluetooth LE roles and states

also indicate whether the device is connectable or non-connectable. In addition to device discovery, advertising packets may also be used for actual data transportation. For example Bluetooth Mesh protocol uses advertising packets as underlying transport bearer [5].

Another Bluetooth LE enabled device acting in Central role scans for advertising packets. When a central device receives connectable advertising from a peripheral device, it may initiate the connection establishment. The device initiating the connection transmits the connection request on the same advertising channel on which it received a connectable advertising packet. If the advertiser accepts the connection, the connection is considered established and the initiator becomes master and the advertising device becomes slave. Once devices are connected, data packets are transmitted on RF channels 0-36 using frequency hopping pattern [5].

2.1.3. Link Layer Protocol

Link Layer protocol is a binary protocol where PDUs are sent over the air on Bluetooth LE Data channels. The LL protocol consists of two different types of PDUs: LL Control and LL Data PDUs. LL PDU format can be seen in Figure 4. Both PDU types share a similar LL header which contains fields telling for example PDU type and total length of PDU. In addition, encrypted PDUs contain Message Integrity Check (MIC) to ensure that message payload has not been tampered and the sender has the shared encryption key [5].

Figure 4. LL PDU format

The Control PDUs form their own protocol called Link Layer Control Protocol (LLCP) which main purpose is to control and negotiate different parameters and aspects of the connection between two Link Layer implementations. For example, there are LL control procedures for features exchange, version exchange, encryption, and termination of connection [5].

The LL Data PDUs are used for transmitting the upper layer protocol data. LL Data PDU consists of the LL header and the payload and optionally MIC. Payload field contains the upper layer data, usually encapsulated into L2CAP packets [5].

2.1.4. Host Controller Interface

HCI is a standardized interface between the host and controller components and it makes it possible to use different controller or host components interchangeable. This is essential as controller implementations are usually tightly coupled with theradio hardware. For example, Linux's Bluetooth stack BlueZ can be used with many hardware options from different vendors. HCI also provides flexibility in terms of supported Bluetooth version: host and controller are not required to support the same Bluetooth version.

The host stack uses HCI commands for controller initialization, configuration, connecting, and for sending data to another device. The controller uses HCI events as a way to pass data to the host stack. HCI events may be used for returning status or return value of the HCI command. They may also be used asynchronously to indicate status changes or for passing data from another device to the host stack.

In addition to standard HCI commands and events, the HCI interface provides a way to use vendor-specific debug commands and events for extending the interface [5].

2.1.5. Bluetooth LE Protocols and Application Profiles

The Bluetooth LE host protocol stack is build on top of Link Layer and HCI which were described earlier in sections 2.1.3 and 2.1.4. On the host stack side, Bluetooth protocols used for actual application use cases are called profiles. The Bluetooth LE protocol stack can be seen in Figure 5. Most of the application protocols and profiles are running on top of L2CAP.

Logical Link Control Adaptation Protocol

Logical Link Control Adaptation Protocol (L2CAP) is a transport protocol between higher layer protocols and lower layer protocols. The L2CAP provides a way to multiplex multiple channels over a single Bluetooth LE link. L2CAP is responsible for flow control and it also provides mechanism for sending packets bigger than the MTU by fragmentation and reassembly [11 p.217].

The L2CAP has channels to separate sequences of packets from different applications. In Bluetooth 4.0, only fixed channels were available meaning that channels were immediately available once the Bluetooth LE link was established. For example, SMP and ATT protocols use fixed channels.

In Bluetooth 4.1, connection oriented channels, also known as LE credit based flow control mode, were introduced to Bluetooth LE. Connection oriented channels made it possible for the receiver to limit the amount of incoming data by flow control. In

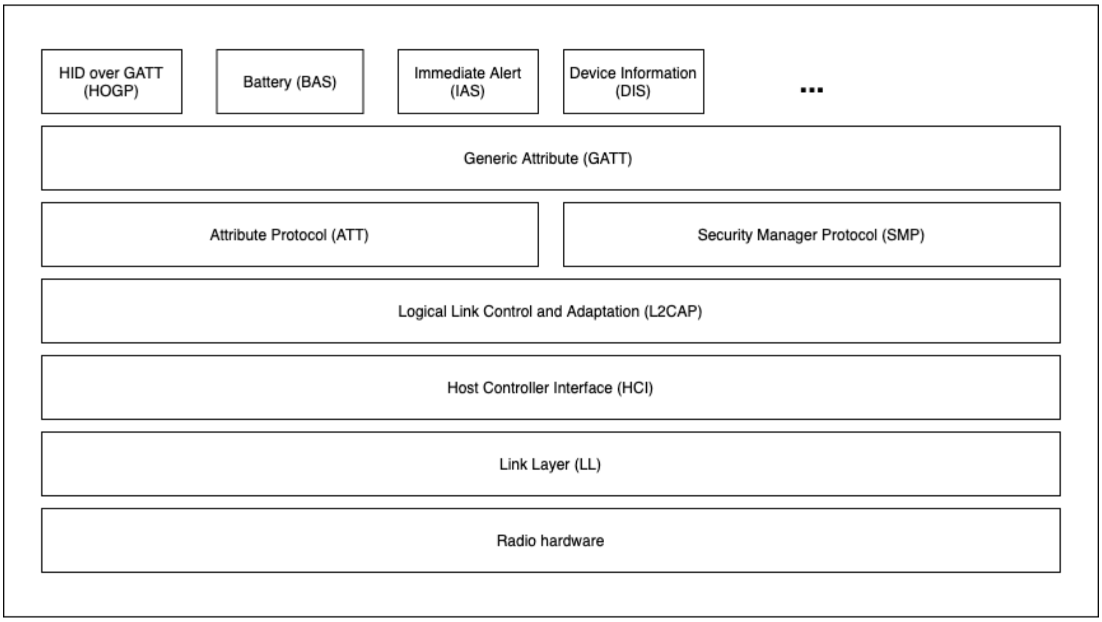

Figure 5. Bluetooth LE protocol stack

this mode, an L2CAP connection is created on a fixed signaling channel. During the creation of the L2CAP connection, L2CAP parameters are negotiated and dynamic channel identifiers are assigned for both ends. Application data is then sent using the channel identifiers [11 p.226].

Bluetooth 5.2 added a new L2CAP mode called enhanced credit based flow control mode. The enhanced credit based flow control mode is very similar to older LE credit based flow control mode. The only difference is that the new mode provides a way to create multiple L2CAP channels on one request. This mode makes it possible to reconfigure L2CAP parameters after the L2CAP connection has been created [5].

Security Manager Protocol

Security Manager Protocol (SMP) defines procedures for pairing, authentication, and encryption between Bluetooth LE devices. The SMP is used when two devices are paired. During the pairing procedure, encryption and identity related keys are generated and distributed. In contrast to Bluetooth BR/EDR, Bluetooth LE SMP does pairing and authentication in the host stack instead of the controller. The Bluetooth 4.2 specification introduced a new feature called Secure Connections which improved the security of Bluetooth LE significantly. The old security mode is called legacy pairing. The SMP utilises L2CAP as transport layer for sending and receiving SMP commands [11 p.231].

Attribute Protocol and Enhanced Attribute Protocol

Attribute protocol (ATT) defines procedures for how to discover, read, and write attributes of another Bluetooth LE device. The attribute represents some kind of data on the device. Every attribute has the following meta data: attribute type, attribute handle, and access permissions [11 p.259].

The ATT uses client-server architecture where one server may serve multiple clients but also a client may connect to multiple servers. The device may also act as client and server at the same time. The server can be imagined as a database where client may write or read attribute's data. The server may also send data to a subscribed client as notifications or indications [11 p.259].

Enhanced Attribute protocol (EATT) was introduced in Bluetooth version 5.2. The EATT improved the ATT by introducing support for concurrent transactions and interleaving of L2CAP packets related to ATT packets from different applications. The EATT also made it possible to change the ATT Maximum Transmission Unit (MTU) during a connection [5].

Generic Attribute Profile

Generic Attribute Profile (GATT) provides a way to describe the hierarchy of GATT based profiles. The GATT is built on top of the ATT and L2CAP protocols and takes the security into account utilising the SMP. The GATT profile consists of one or more GATT Services. Every GATT service is a collection of GATT Characteristics which in turn represents attributes in ATT terminology [11 p.285].

GATT based profiles and services

Even though it technically would be possible to design a Bluetooth LE profile directly on top of L2CAP, most of the Bluetooth LE application profiles are based on the GATT. Both iOS and Android provide a GATT based API for application developers and do not provide direct access to L2CAP API [12, 13]. There are lots of GATT based application profiles. A few of them are described below as an example.

- **Hid over GATT Profile (HoGP)** is traditional Human Interface Device (HID) protocol converted to GATT profile.
- **Batter Service (BAS)** provides battery level information of remote device as notifications.
- **Immediate Alert Service (IAS)** provides way to execute certain action immediately in remote device, for example blink led or play sound.
- **Device Information Service (DIS)** provides information about manufacturer of the remote device.

Internet Protocol Support Profile

Internet Protocol Support Profile (IPSP) is one of the very rare non-GATT based profiles. IPSP uses L2CAP LE credit based flow control mode or enhanced credit based flow control mode as a transport and provides mechanism to exchange IPv6 packets between two Bluetooth LE enabled devices. IPSP has two different kind of devices: nodes are edge devices and routers are devices which have access to internet. The routers are acting as a gateway for nodes and they do not need to have any knowledge on the type of data. IPSP is typically used in IoT systems where some kind of sensor needs to be able to send data to internet via a smartphone or some other acting as an IPSP router [11 p.371].

2.2. Software Testing

As the software is composed by humans, there is no software without errors, also called bugs. Traditionally, software testing has been defined as a process of trying to find those bugs [3]. However, nowadays software testing covers not only bug hunting but also verification and validation. As per the IEEE's definition, verification is the process of evaluating a system or component to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase [14]. Validation in turn has been defined as the process of evaluating a system or component during or at the end of the development process to determine whether it satisfies specified requirements [14]. Those two explanations can be simplified to two questions: "Did we build the product right?" and "Did we build the right product?" [4 p.13]. However, testing is never able to fully prove that there are no bugs in software. The purpose of testing is to show that faults are present [15 p.14].

2.2.1. Black-Box and White-Box Testing

Software testing can be categorized into two different strategies: black-box testing and white-box testing. In black-box testing, the system under test is viewed as a black-box which does not reveal any information about the inte§rnals of the system. This means that there is no prior knowledge of internal structure of the system and the test target is approached through its interfaces such as command line, graphical user interface, network protocols, files, or APIs [4 p.85].

White-box testing strategy in turn is different because it is based on assumption of having access to source code. One good example of white-box testing are automatic static analysis tools [4 p.83].

2.3. Fuzz Testing

With computers and computer-like devices being exposed to wireless and wired networks, nowadays devices are expected to be able to handle invalid or otherwise unexpected input. Fuzz testing, also known as fuzzing, is a black-box software testing methodology which consists of providing intentionally malformed input to a system under test (SUT) and observing the behaviour of the SUT. In the context of software testing, black-box testing means that input is delivered to the target software through communication interfaces with little or no information about the internals of the system under test [4 p.26].

2.3.1. History of Fuzzing

Before fuzzing was invented in the software world, negative testing was used for hardware testing already decades ago [4]. In 1990, Barton Miller, Lars Fredriksen, and Bryan So developed a tool called fuzz which generated stream of random characters. An inspiration for their research was the observation that modem line noise caused crashes in certain UNIX utilities. They ran the fuzz testing program against many different UNIX utility programs and found out that 24% of tested utility programs crashed to malformed input [16].

Inspired by the research of Miller et. al., Oulu Secure Programming Group (OUSPG) started the PROTOS project in 1999. Later, OUSPG and Technical Research Center of Finland (VTT) presented results of the PROTOS project which introduced new approaches for testing protocol implementations using black-box methods. As a result of the project, researchers could identify many serious vulnerabilities in ASN.1 parsers [17, 4 p.23].

In the beginning of the 2000s, the first commercial fuzzing solutions started to appear. Also a few open-source fuzzers were published. At that time, most of the fuzzers were based on mutation of input sample files or grammar. In 2014, Michal Zalewski published American Fuzzy Lop (AFL) which provided a huge improvement in effectiveness of fuzzers. AFL used compile-time binary instrumentation for providing feedback for test case creation. Feedback-based fuzzing was very effective in finding new execution paths which in turn revealed bugs very quickly. Inspired by AFL, many new coverage guided fuzzers have been introduced in the late 2010s [4 p.23]. Still after 30 years of fuzzing there seems to be room for improvement: standard operating system utilities are still crashing for very similar reasons as 30 years ago [18].

2.3.2. Different Categories of Fuzzing

There are many ways to categorize fuzzers based on their capabilities. Takanen et. al. present two ways to categorize fuzzers [4 p.28]:

- How test cases are delivered to System Under Test (SUT)
- Test case complexity

Fuzz test cases can be delivered to target software over different mediums. Some fuzzers are able to transfer test cases over traditional network protocols like Ethernet, Internet Procotol (IP), Transmission Control Protocol (TCP), User Datagram Protocol (UDP), and Stream Control Transmission Protocol (SCTP) and are targeting higher level protocol implementations on top of those transport protocols. There are also fuzzers for wireless protocols like Wi-Fi and Bluetooth. File format fuzzers can be used for fuzzing applications or libraries taking files as an input [4 p.28].

Another way to categorize fuzzers is based on the test case complexity. Template-based fuzzers use some kind of template as a base for test case generation. They do not have any kind of deeper knowledge of the protocol or file format and they typically are not able to resolve the dynamic contents of protocols, for example, checksums. The template can be, for example, network traffic capture of a sample file. Block-based fuzzers are improved version of template-based fuzzers: they are able to cover the dynamic functionality of protocols such as checksums and length fields [4 p.30].

Evolution-based fuzzers use a feedback loop for learning which test case is effective and generate new test cases based on that feedback. Evolutionary fuzzers may be able to learn the protocol or file format without having prior knowledge about the specification [4 p.30].

Model-based fuzzers generate the test cases based on the protocol model. These fuzzers look like valid protocol implementation as they are able to properly handle the dynamics of protocol or file format. They typically are able to calculate lengths and checksums and do cryptographic operations correctly which makes them effective as the test cases are not rejected in the very first checksum or length field check when the SUT receives them. In addition to trying out invalid values for all of the message fields, model-based fuzzers also try all the alternative valid values [4 p.30].

2.4. Synopsys Defensics

Synopsys Defensics [19] is a commercial model-based protocol fuzzing framework including test suites for over 250 different network protocols and file formats. In addition to traditional network protocols, Defensics is capable of fuzzing wireless protocols like Bluetooth and Wi-Fi. All the test suites consist of comprehensive model of protocol, anomalizer engine and the injector component which takes care of transporting the test case to target.

Codenomicon, the company originally developing Defensics, was founded in 2001 and acquired by Synopsys in 2015. Codenomicon was one of the first companies selling fuzz testing tools. Originally, Codenomicon was spin-off from OUSPG's PROTOS research project.

In 2014, the team behind Synopsys Defensics discovered the infamous Heartbleed vulnerability in one of the most popular open source cryptographic library OpenSSL [20]. The vulnerability allowed an attacker to steal secret keys or credentials without leaving any traces on the server. As serious as the Heartbleed vulnerability was, it has had even bigger impact on the way security vulnerabilities have been published during the last few years.

3. RELATED WORK

3.1. Wireless Security Evaluation Frameworks

Many different wireless security evaluation frameworks have been proposed and evaluated by research community. Even though this thesis mainly focuses on Bluetooth LE, also security evaluation frameworks for other wireless technologies were studied.

The wireless security evaluation frameworks described in this chapter can be categorised based on their features. Table 1 shows the differences between Ubertooth, InternalBlue and Greyhound. In the context of this table, passive monitoring means capability of monitoring Bluetooth BR/EDR or Bluetooth LE connectionless and connected communication. Link layer packet reception and injection mean the capability of sending arbitrary Bluetooth BR/EDR Link Manager Protocol (LMP) or Bluetooth LE LL packets to target device and capability to receive responses from the target device. Capability to be used as standard Bluetooth device means that the device could be used as a hardware for other Bluetooth BR/EDR or LE security evaluation tools which are more targeted for testing host stack and not so much dependent on specific hardware.

Table 1. Capabilities of security evaluation frameworks

Capability	Ubertooth	InternalBlue	Greyhound
Passive monitoring	yes	no	no
Link layer packet reception	yes	yes	yes
Link layer packet injection	no	yes	yes
Usable as standard Bluetooth device	no	no	no

3.1.1. Ubertooth

Ubertooth One is a open source tool which can be used for monitoring Bluetooth LE and Bluetooth BR/EDR communication over the air [21]. It was originally introduced by Michael Ossman at ShmooCon in 2011 and it was the first free option for commercial Bluetooth sniffers like Teledyne Lecroy or Ellisys. Later Nordic Semiconductor has released free nRF sniffer for capturing Bluetooth LE and 802.15.4 protocol communication [22].

The Ubertooth is able capture and demodulate 2.4 GHz signals which include for example Bluetooth BR/EDR and Low Energy. It consists of open source hardware, firmware and host code. The Ubertooth One hardware is built from following main components: NXP LPC175x microcontroller, TI CC2400 wireless microcontroller and TI CC2591 RF front end. The host code and firmware communicate through USB interface using custom serial over USB communication protocol. [21]

Ubertooth has been used successfully in many research projects [23, 24, 25]. Even though Ubertooth One is very useful for monitoring the Bluetooth LE traffic, it doesn't support packet injection.

3.1.2. InternalBlue

InternalBlue is a open source Bluetooth security evaluation platform mainly targeted for security researchers. It is capable of injecting and receiving arbitrary Bluetooth BR/EDR LMP messages on specific Broadcom chipsets. It uses Broadcom Bluetooth controller which is embedded in Google Nexus 5 and provides way to receive and inject arbitrary Bluetooth BR/EDR LMP packets. Researchers reverse-engineered firmware of Broadcom Bluetooth development kit and found a proprietary diagnostics functionality. They reverse-engineered the diagnostic protocol that relied on Broadcom's vendor specific HCI commands. Using Android Debug Bridge (ADB) they were able to use the diagnostic functionality for sniffing Bluetooth BR/EDR traffic and sending and receiving LMP packets on off-the-shelf Nexus 5 phone [26].

InternalBlue was introduced in 2018 by Dennis Mantz in his Master's thesis. Later more features has been added and it has been covered in many research papers and conference talks by researchers of Secure Mobile Networking Lab of Technical University of Darmstadt [1, 27].

InternalBlue has been an inspiration for many other related researches, including Greyhound [28] and this thesis.

3.1.3. Greyhound

Greyhound is a automated fuzzing framework for discovering vulnerabilities in Bluetooth LE and Wifi implementations. Mutation based fuzzer uses systematic approach to optimize the mutations using particle swarm optimization (PSO). In addition to sending malformed input to target device, Greyhound tries to find state handling issues by sending valid looking input in wrong protocol state. [29, 28]

The Bluetooth LE solution of Greyhound consists of Nordic nRF52840 based Bluetooth LE dongle running customized firmware and separate fuzzer application running on PC. The dongle bypasses HCI interface fully and provides proprietary API for injecting fuzzed Bluetooth LE packets over the air. During the research done by M. E. Garbelini et. al., Greyhound was used for fuzzing 12 different Bluetooth LE devices from ten vendors. In total researchers discovered total of 17 different vulnerabilities which got named Sweyntooth [29].

The Wifi solution of Greyhound consists of off the shelf Ralink RT3070 dongle, customized device driver and the fuzzer application [28].

3.2. Bluetooth Vulnerabilities

During the last few years, many Bluetooth related vulnerabilities have been discovered and reported by security community. Vulnerabilities affecting deeply embedded operating systems may be attractive from attacker's point of view because most of the deeply embedded operating systems are missing basic exploit mitigation techniques like Executable Space Protection (ESP), Address Space Layout Randomization (ASLR) and stack canaries [30]. Many recent Bluetooth related vulnerabilities have impacted implementations of lowest protocol layers of Bluetooth in a firmware running on separate embedded System on Chip dedicated for wireless communication. Few of the most significant recent Bluetooth vulnerabilities affecting Bluetooth link layer are described in this chapter.

3.2.1. Sweyntooth

Sweyntooth is a code name for set of 17 different vulnerabilities discovered and disclosed by Singapore University of Technology and Design researchers [29]. The researchers used the Greyhound fuzzer described in section 3.1.3 in their study. The vulnerabilities were discovered by fuzzing different vendor's Bluetooth LE implementations on respective development kits. The consequences of vulnerabilities were demonstrated against five different IoT end products which were using vulnerable software development kits (SDK). These vulnerabilities were found in various Bluetooth LE SDKs from 10 different vendors and open source projects. According to the researchers, the vulnerable SDKs have been used in over 480 end products.

Sweyntooth vulnerabilities were classified to three different types [29]:

- **Crash:** Attacker could crash the target device by sending malformed Bluetooth LE packet. Often the cause of this kind of vulnerability is memory corruption. Memory corruption vulnerabilities are very likely exploitable and could lead to remote code execution.
- **Deadlock:** Attacker could cause Denial of Service (DoS) situation by sending malformed Bluetooth LE packet to target device. Target device does not crash but goes otherwise in non-functional state.
- **Security bypass:** Attacker could bypass security measures of target device fully or partially.

Table 2 lists all the Sweyntooth vulnerabilities and categorizes them based on the affected part of the Bluetooth LE stack. Total 11 out of the 17 Sweyntooth vulnerabilities affect the Link Layer implementations.

Table 2. Sweyntooth vulnerabilities

CVE	Description	Affected part of stack
CVE-2019-17061	Link Layer LLID deadlock	Link Layer
CVE-2019-17060	Link Layer LLID deadlock	Link Layer
CVE-2019-17517	Truncated L2CAP	Link Layer
CVE-2019-17518	Silent Length Overflow	Link Layer
CVE-2019-19193	Invalid Connection Request	Link Layer
CVE-2019-17520	Unexpected Public Key Crash	Host
CVE-2019-19192	Sequential ATT Deadlock	Host
CVE-2019-19195	Invalid L2CAP fragment	Link Layer
CVE-2019-19196	Key Size Overflow	Host
CVE-2019-19194	Zero LTK Installation	Link Layer
CVE-2020-13593	DHCheck Skip	Host
CVE-2020-13595	HCI Desync Deadlock	Link Layer
CVE-2020-10061	Invalid Sequence Memory Corruption	Link Layer
CVE-2020-10069	Invalid Channel Map Crash/Deadlock	Link Layer
CVE-2020-13594	Invalid Channel Map Crash/Deadlock	Link Layer

3.2.2. KNOB

Key Negotiation of Bluetooth (KNOB) attack exploits a weakness in the Bluetooth specification. Bluetooth BR/EDR security specification includes an encryption key negotiation protocol which is used when negotiating encryption keys. Due to the weakness in the specification, attacker can remotely manipulate the entropy negotiation and force the Bluetooth devices to use key with only one byte entropy and then brute force the keys in real time. The KNOB attack was assigned CVE-2019-9506 [31].

Bluetooth BR/EDR security features are implemented fully in Link Manager Protocol (LMP) in the controller. This means that KNOB attack targets the firmware of Bluetooth chips. The attack is very effective as it exploits the weakness in specification. In practise this means that all Bluetooth BR/EDR can be expected to be vulnerable. The attack is also stealthy as Bluetooth BR/EDR controller does not notify the host stack about the key negotiation. User has no way to notice that encryption key has been cracked and confidentiality of the Bluetooth connection lost [32].

Researchers who discovered the vulnerability in specification, implemented proof of concept using the InternalBlue [33] Bluetooth security evaluation framework. The InternalBlue has been described in section 3.1.2.

4. DESIGN AND IMPLEMENTATION

The main goal of this thesis was to design and implement a solution for injecting arbitrary Bluetooth LE Link Layer PDUs over the air. The solution should be tested and verified using a test suite developed using Synopsys Defensics Software Development Kit (SDK) [19]. Testing should involve sending and receiving different kind of arbitrary Link Layer PDUs to a target device and monitoring the Bluetooth LE packets over the air using nRF sniffer [22] and Wireshark [34].

4.1. System Description

On a high level, the Bluetooth LE Link Layer fuzzing solution comprises of two main components: host stack and controller. In the scope of this thesis, the host stack also includes the fuzz test suite used for generating fuzz test cases for the protocol. The test Suite and host stack run on a regular computer running Linux operating system. The controller component runs on Bluetooth LE USB dongle hardware.

4.2. Choice of Hardware and Firmware

Even though there are many Bluetooth LE dongles on the market, any off-the-shelf Bluetooth LE product does not allow manipulating and sending raw Link Layer PDUs over the air. Therefore, the selected hardware and firmware need to fulfill certain requirements. Both hardware and firmware decisions are depending on each other: selecting certain hardware limits the choices of firmware and vice versa.

4.2.1. Firmware Selection Criteria

To be able to design and implement a system that allows manipulating Bluetooth LE Link Layer PDUs, custom firmware was needed. There were three different open source Bluetooth LE controller implementations available: Zephyr [35], Mynewt NimBLE [8], and Packetcraft [10]. The decision between these three was made based on the following criteria:

- Supported hardware
- Availability of documentation
- Community activity
- License

The first criterion for firmware selection was hardware support. Firmware should support at least one of the identified hardware options. It was clear already from the beginning that firmware needs to be customized and using public APIs is not enough. Documentation should be comprehensive and not limited to public API documentation. The availability of documentation was graded with scale "non-existent", "sufficient", and "excellent". In addition to documentation, an active and participating community was seen asa benefit. Also community activity was graded with scale "non-existent", "sufficient", and "excellent". Finally, a permissive open source software license was preferred. Table 3 presents the comparison between firmware options using the criteria and grading described earlier.

Table 3. Comparison between firmware options

Criteria	Zephyr OS	Mynewt Nimble	Packetcraft
Supported hardware	2/2	2/2	0/2
Availability of documentation	excellent	sufficient	non-existent
Community activity	excellent	sufficient	non-existent
License	Apache-2.0	Apache-2.0	Apache-2.0

Based on the firmware selection criteria, Zephyr OS was selected to be the basis for the custom firmware of the solution.

4.2.2. Hardware Selection Criteria

The purpose of this project was not to do actual hardware design but to find an off-the-shelf hardware product that would allow developing Bluetooth LE dongle using customized firmware. Available firmware choices limited the available hardware choices quite a bit as the only chipset supported by all of the firmware options was Nordic nRF52840. Two different suitable hardware options were identified on the market: Laird BL654 and Fanstel USB840F. The following criteria were defined for the hardware choice decision:

- Hardware certifications
- Form factor
- Availability and pricing

The most important aspect when selecting hardware was that it needed to be in a form that it is ready to be shipped to customers world wide. Certain level of hardware certifications were seen necessary to ensure that devices can be used globally.

Form factor was important criteria from two different points of view: In addition to being usable by the customer, it should be easy to trigger the device into bootloader mode to enable initial flashing before shipping to customers. A USB dongle was the preferred format for hardware.

Availability and pricing of the hardware was also seen as an important aspect: the selected hardware should be commonly used and well available from different distributors for a reasonable price. Table 4 presents the comparison between hardware options.

Table 4. Comparison between hardware options

Criteria	Laird BL654	Fanstel USB840F
Hardware certification	FCC, IC, CE, MIC and RCM	FCC, IC, CE, MIC and RCM
Form factor	USB dongle	USB dongle[1]
Availability and pricing	8 global distributors[2]	3 global distributors[2]

Based on the hardware selection criteria, Laird BL654 was selected to be the hardware component of the solution.

4.2.3. Firmware: Zephyr OS

The Zephyr OS is an Apache licensed real-time operating system project backed by the Linux Foundation and many big industry vendors like Intel, Nordic Semiconductor, Google, Facebook, and Synopsys [35]. It is mainly targeted to embedded and resource constrained systems. Zephyr supports very wide variety of boards with different CPU architectures including ARM Cortex-M, Intel x86, ARC, NIOS II, Tensilica Xtensa, SPARC V8 and RISC-V 32. Zephyr's native network stack supports multiple protocols including LwM2M, BSD sockets, OpenThread, full Bluetooth LE stack including Link Layer and Bluetooth Mesh [36].

The Zephyr project has a very active community and comprehensive documentation. Zephyr's governance model is divided into two governing groups: administrative and technical. Member organizations may participate in both governing bodies and take part in project's decision making and technical direction [35].

4.2.4. Hardware: Laird BL654

Laird BL654 is a nRF52840 System on Chip (SoC) based device in USB dongle form. It includes 2.4GHz radio which can be used for many different wireless use cases like Bluetooth LE, Zigbee, and Thread. The dongle has a wide variety of hardware certifications: FCC, IC, CE, MIC, and RCM [37]. The dongle is available from multiple distributors with high reserves.

4.3. Option 1: Link Layer PDU Manipulation

Initially, two different approaches were identified for injecting arbitrary LL PDUs. FThe first option for LL injection was LL PDU manipulation in the controller right before the PDU is passed to radio and sent over the air. In this approach, Link Layer state handling stays fully in the controller.

[1]Resetting to bootloader mode requires opening the enclosure
[2]As of January 2021

4.3.1. Host Stack and Test Suite

The host stack's role in LL PDU manipulation is to take care of controller initialization, starting and stopping scanning, and initiating a connection. In addition, the host stack must provide a way to send custom HCI commands. A third-party Bluetooth stack was used for this purpose.

The test suite's role is to automatically generate valid and invalid Bluetooth LE LL PDUs based on the model of the Bluetooth LE LL protocol and using the host stack's APIs send and receive LL PDUs. The test suite was implemented using Synopsys Defensics Fuzz Testing Software Development Kit. Design and implementation of the test suite is outside of the scope of this thesis and therefore not described in detail here.

4.3.2. Extended HCI Interface

To be able to manipulate LL PDUs, a standard HCI interface was extended with vendor-specific commands and events.

A set LL Injection command is used for enabling raw LL injection. This also disables LL control procedure state handling in the controller with an assumption that state is handled in the host. Enabling a raw LL injection also enables LL tracing which passes all the TX/RX LL PDUs to the host using vendor-specific HCI events. See: Zephyr Trace Information Event.

```
+-------------------------+-------+--------------------+--------------------+
| Command                 | OCF   | Command            | Return             |
|                         |       | Parameters         | Parameters         |
+-------------------------+-------+--------------------+--------------------+
| Set_LL_Injection        | 0x212 | Enable             | Status             |
+-------------------------+-------+--------------------+--------------------+
```

Figure 6. Format of the Set LL Injection HCI command

Manipulate Link Layer PDU commands provides a way to send malformed Link Layer Control PDUs instead of valid ones. The format of the command can be seen in Figure 7. When this command is used, the controller takes care of state handling. To be able to replace valid PDU with malformed, this command must be called before the connected state.

```
+-------------------------+-------+--------------------+--------------------+
| Command                 | OCF   | Command            | Return             |
|                         |       | Parameters         | Parameters         |
+-------------------------+-------+--------------------+--------------------+
| Manipulate_LL_PDU       | 0x211 | Opcode,            | Status             |
|                         |       | Header,            |                    |
|                         |       | Opcode anomaly,    |                    |
|                         |       | Length,            |                    |
|                         |       | Data               |                    |
+-------------------------+-------+--------------------+--------------------+
```

Figure 7. Format of the Manipulate LL PDU HCI command

The command parameter Opcode tells the controller what LL PDU should be replaced with the one provided in this command. Header, Opcode anomaly, Length, and Data parameters include the actual content to be sent instead of the original PDU.

By default, the Bluetooth host stack does not have capability to sniff LL PDUs. The Zephyr project has specified a format for Zephyr Trace Information vendor-specific HCI events [38]. It provides the format for delivering LL PDUs to the host stack as HCI events for tracing purposes. The format of the Zephyr Trace Information Event in Figure 8.

```
+------------------------------+------------+------------------------------+
| Event                        | Event Code | Event Parameters             |
+------------------------------+------------+------------------------------+
| Trace_Information            | 0xFF       | Subevent_Code,               |
|                              |            | Trace_Type,                  |
|                              |            | Trace_Data                   |
+------------------------------+------------+------------------------------+
```

Figure 8. Format of the Zephyr Trace Information HCI Event

The Zephyr project has defined a specification for vendor-specific HCI extension for LL tracing purposes. The Zephyr Trace Information Event was used for delivering LL PDUs received over the air to the host stack and the test suite.

4.3.3. Controller

The Zephyr Bluetooth LE controller was altered so that the controller holds a buffer containing the information about the opcode of the next PDU to be replaced. The buffer also contains the fuzzed PDU that is going to replace the original one and is going to be sent over the air. Simple algorithm (Algorithm 1) was utilised for all the desired control procedures and for connection establishment for checking whether the original or anomalised PDU was to be transmitted.

Algorithm 1. Replace LL PDU

if *anomaly for PDU with opcode X pending* **then**
    transmit(anomalized);
else
    transmit(original);
end

Even though the Zephyr project already included the specification for Trace Information events, it was missing the implementation for link layer tracing. The link layer tracing using Trace Information vendor-specific HCI events was implemented as it was used for passing transmitted and received LL PDUs up to the host stack.

4.4. Option 2: Raw Link Layer Injection

The second approach for injecting arbitrary LL PDUs was to move the LL state machine to the test suite running on host and provide raw LL injection API as a vendor specific HCI extension. In practise. the test suite tells the controller to send a LL PDU using the vendor-specific HCI command. All the incoming LL PDUs the controller receives are passed as HCI events to the test suite.

4.4.1. Host Stack and Test Suite

Similar to the LL PDU manipulation in section 4.3.1, the host stack's role in raw LL PDU injections is to take care of controller initialization, starting and stopping scanning, initiating a connection, and to provide a way to send custom HCI commands. The same host stack was used also with this option.

In addition to the test suite's role mentioned in section 4.3.1, in case of raw LL injections, the test suite must take care of state handling of the LL protocol. This involves, for example, initiating control procedures, handling of remote initiated control procedures, encryption, and connection termination.

4.4.2. Extended HCI Interface

The extension for the HCI interface was needed to be able to send arbitrary Link Layer PDUs over the air.

The same Set LL Injection HCI command was used also with this approach. The command disables the LL control procedure state handling in the controller with an assumption that state is handled in the host. The format of Set LL Injection HCI commands can be seen in Figure 6. Enabling raw LL injection also enables LL tracing which passes all the transmitted and received LL PDUs to thehost using the Zephyr Trace Information HCI event seen in Figure 8.

The Send Raw LL PDU command seen in Figure 9 provides a way to send arbitrary Link Layer Control and Data PDUs over the air. The device must be in a connected state to be able to send the PDU to the target device. To be able to use the Send Raw LL PDU HCI command, the LL injection must have been enabled using the Set LL Injection HCI command described earlier in this chapter.

```
+-------------------------+-------+--------------------+--------------------+
| Command                 | OCF   | Command            | Return             |
|                         |       | Parameters         | Parameters         |
+-------------------------+-------+--------------------+--------------------+
| Send_Raw_LL_PDU         | 0x210 | Handle,            | Status             |
|                         |       | Header,            |                    |
|                         |       | Length,            |                    |
|                         |       | PDU                |                    |
+-------------------------+-------+--------------------+--------------------+
```

Figure 9. Send Raw Link Layer PDU HCI command

4.4.3. Controller

This approach required wider changes to the Zephyr Bluetooth LE controller than the first one. The same method was used for incoming LL PDUs: they were passed to the host stack using the Zephyr Trace Information HCI Event shown in Figure 8. All the LL control procedure handling was changed so that enabling LL injections disabled the state handling in the controller. Some of the control procedures, for example, Features exchange, were originally initiated automatically right after connection was established. Al these also had to be disabled when LL injection was enabled.

The implementation of sending raw LL PDUs itself was relatively simple. By default, the Zephyr Bluetooth LE controller uses two different packet queues for outgoing LL PDUs: one for control PDUs and another for Data PDUs. The Bluetooth LE controller was modified so that when the controller receives a Send Raw LL PDU HCI command, it places the PDU directly to either the control PDU queue or data PDU queue depending on the type of the PDU.

5. EXPERIMENTS

Two different solutions were designed, implemented, and tested in practise for Bluetooth LE LL injections. Both approaches were tested by sending arbitrary Link Layer PDUs to the target device. Nordic nRF52840 development kit running Zephyr Bluetooth shell application was used as a test target. Bluetooth LE packets were captured over the air using nRF sniffer [22] and Wireshark [34].

Both approaches had their advantages and disadvantages. A comparison of options 1 and 2 and their capabilities can be seen in Table 5.

Table 5. Comparison of two different LL injection options

Capability	Option 1	Option 2
Send arbitrary LL ADV PDU on Advertising channels	X	
Send single arbitrary LL Control PDU on Data channels	X	X
Send sequence of arbitrary LL Control PDUs on Data channels		X
Receive LL PDUs on Data channels	X	X
State anomalies		X

5.1. Option 1: Link Layer PDU Manipulation Experiments

The main benefit of the first option was the simplicity of the test suite side implementation. The LL state machine stayed in the controller which meant that the test suite didn't need to implement the Link Layer state machine and, for example, encryption and decryption routines. With the design choices made, the LL PDU manipulation approach made it possible to manipulate basically any LL PDU including Advertising and Data PDUs but only one PDU at time. The unique capability of this approach was that it made it possible to manipulate also Advertising PDUs like CONNECT_IND. Figure 10 shows malformed CONNECT_IND ADV PDU which was captured from the air.

```
▼ Bluetooth Low Energy Link Layer
    Access Address: 0x8e89bed6
  ▼ Packet Header: 0x22e5 (PDU Type: CONNECT_IND, ChSel: #2, TxAdd: Random, RxAdd: Random)
      .... 0101 = PDU Type: 0x5 CONNECT_IND
      ...0 .... = Reserved: 0
      ..1. .... = Channel Selection Algorithm: #2
      .1.. .... = Tx Address: Random
      1... .... = Rx Address: Random
      Length: 34
    Initiator Address: WesternD_00:00:01 (c0:00:00:00:00:01)
    Advertising Address: 7f:22:c4:6e:2d:1c (7f:22:c4:6e:2d:1c)
  ▼ Link Layer Data
      Access Address: 0x55a1329d
      CRC Init: 0xad503c
      Window Size: 1 (1,25 msec)
      Window Offset: 43690 (54612,5 msec)
      Interval: 80 (100 msec)
      Latency: 0
      Timeout: 500 (5000 msec)
    ▶ Channel Map: ffffffff1f
      ...0 1101 = Hop: 13
      001. .... = Sleep Clock Accuracy: 151 ppm to 250 ppm (1)
    CRC: 0xde364e

0000  78 35 00 02 06 9a 06 0a  01 25 21 00 00 96 00 00   x5······ ·%!
0010  00 d6 be 89 8e e5 22 01  00 00 00 00 c0 1c 2d 6e   ······"· ·······-
0020  c4 22 7f 9d 32 a1 55 3c  50 ad 01 aa aa 50 00 00   ·"··2·U<
0030  00 f4 01 ff ff ff ff 1f  2d 7b 6c 72               ········ -{lr
```

Figure 10. Malformed CONNECT_IND PDU captured from the air

The Link Layer PDU manipulation method can also be used for sending arbitrary LL Control PDUs on Data channels with the expectation that the controller can be driven into a state where the wanted PDU would be sent. In practise, this requires that the test suite uses standard HCI commands for requesting certain LL procedures. Figure 11 shows malformed LL_FEATURE_REQ PDU captured from the air.

```
▼ Bluetooth Low Energy Link Layer
    Access Address: 0xea35451e
    [Master Address: d6:65:24:4b:97:97 (d6:65:24:4b:97:97)]
    [Slave Address: 58:23:e2:71:7e:cc (58:23:e2:71:7e:cc)]
  ▶ Data Header: 0x090f
    Control Opcode: LL_FEATURE_REQ (0x08)
  ▶ Feature Set: 0xaaaaaaaaaaaaaaaa
    CRC: 0x1f8bcf

0000  78 1c 00 02 1c 9c 06 0a  03 0c 24 01 00 66 85 01   x······· ··
0010  00 1e 45 35 ea 0f 09 08  aa aa aa aa aa aa aa aa   ··E5····
0020  f8 d1 f3                                           ···
```

Figure 11. Malformed LL_FEATURE_REQ PDU captured from the air

The downside of this approach was that only one PDU per connection could be manipulated reliably. This approach did not allow anomalizing sequences of PDUs. Also, it appeared that this approach limited the coverage of LL Control Procedures that could be anomalized when comparing to the second approach.

Common in both options is that they are capable of receiving LL PDUs and passing them through the HCI interface and host stack up to the test suite.

5.2. Option 2: Raw Link Layer Injection Experiments

In the second approach, the raw LL injection was provided as a vendor-specific extension to the HCI interface. The LL state machine was modeled into the test suite which meant that test suite implementation got more complicated but also provided a very important benefit: it made it possible to fuzz the state handling of the target device. In practise, this means that the test suite is capable of sending valid looking PDUs in wrong state, sending duplicate PDUs, and building complex sequences of PDUs. Also encryption and decryption were implemented in the test suite which provides a way to anomalize the encryption procedure. Figure 12 shows that raw LL injection method is capable of sending valid looking LL PDUs in wrong state.

```
d6:65:24:4b:97:97  LE 1M  LE LL   34 150µs                            0 CONNECT_IND
Master_0xb58d8b55  LE 1M  LE LL    0 1238µs     0     0     False     0 Empty PDU
Slave_0xb58d8b55   LE 1M  LE LL    0 150µs      0     1     False     0 Empty PDU
Master_0xb58d8b55  LE 1M  LE LL    3 99686µs    1     1     False     1 Control Opcode: LL_PHY_REQ
Slave_0xb58d8b55   LE 1M  LE LL    9 150µs      1     0     False     1 Control Opcode: LL_SLAVE_FEATURE_REQ
Master_0xb58d8b55  LE 1M  LE LL    9 99590µs    0     0     False     2 Control Opcode: LL_FEATURE_RSP
Slave_0xb58d8b55   LE 1M  LE LL    3 150µs      0     1     False     2 Control Opcode: LL_PHY_RSP
Master_0xb58d8b55  LE 1M  LE LL    9 99589µs    1     1     False     3 Control Opcode: LL_LENGTH_REQ
Slave_0xb58d8b55   LE 1M  LE LL    6 151µs      1     0     False     3 Control Opcode: LL_VERSION_IND
Master_0xb58d8b55  LE 1M  LE LL    5 150µs      0     0      True     3 Control Opcode: LL_PHY_UPDATE_IND
Slave_0xb58d8b55   LE 1M  LE LL    0 151µs      0     1     False     3 Empty PDU
Master_0xb58d8b55  LE 1M  LE LL    2 150µs      1     1     False     3 Control Opcode: LL_TERMINATE_IND
```

Figure 12. LL_LENGTH_REQ PDU sent in illegal state during PHY update procedure

The main weakness of the second option was that it required that devices are in connected state which meant that it was not possible to send invalid Advertising PDUs like CONNECT_IND over the air.

5.3. Results

The goal of this thesis was to enable Bluetooth LE LL fuzzing by designing and implementing a novel way to inject arbitrary LL packets over the air. Two different approaches for Bluetooth LE LL injection were designed and implemented. Based on the experiments of those two approaches, both have their clear advantages and disadvantages. However, when combining both approaches to one solution, it is fair to claim that the goal of the thesis was achieved.

5.4. Discussion

To evaluate the capabilities of the developed solution in the bigger picture, a comparison was made against prior research. Greyhound [29] and InternalBlue [26] were used as a baseline for comparison as they were the only available existing solutions that were able to provide arbitrary packet injection on the lowest layer of Bluetooth. Because comparison of actual fuzzer components was out of the scope of this thesis, only packet injection methods were evaluated.

InternalBlue, the original inspiration of this thesis, deserves special credit for being the first solution providing a way to fuzz test the lowest layers of Bluetooth. InternalBlue is more suitable for academic research or hacker-minded people, whereas our solution's target personas are cyber security or quality assurance professionals looking for commercial grade Bluetooth fuzzing tools. Researches behind InternalBlue used Broadcom's proprietary Bluetooth controller included in off the shelf Nexus 5 smart phones, binary patched the Bluetooth controller firmware which obviously was not a viable approach for us developing a commercial product. Common between our solution and InternalBlue was that both utilise vendor-specific extensions of the HCI interface for passing lower layer packets between the host stack and controller for injection and tracing purposes. Because InternalBlue supports Bluetooth BR/EDR and our solution is Bluetooth LE only, it is impossible to fully compare all the capabilities the injection method provides.

Greyhound's Bluetooth LE fuzzing solution is hardware-wise very close to our solution. It also uses Nordic's nRF52840 Bluetooth LE USB dongle for injecting the packets. The main difference between these two solutions is that Greyhound provides a proprietary host API instead of relying on the standard HCI interface and vendor-specific extensions. Greyhound's fuzzer communicates directly with the controller over UART. The main benefit of our approach with using a standard HCI interface with extensions is that our solution can be used also as a regular Bluetooth LE dongle on standard Linux machine. This makes it possible to use same hardware with other Bluetooth security evaluation tools and fuzz test suites.

In general, during the last 30 years, fuzzing techniques have evolved a lot. All started from a simple programa called fuzz which was able to output random characters and cause crashes in operating system utilities. Nowadays, the best-in-class fuzzers use advanced instrumentation methods for guiding the test case generator to optimise the effectiveness of fuzz testing. Still there are a few challenges to be solved to make sure fuzzing will keep improving the security and robustness of software also in future.

To enable fuzzing earlier in the software development life cycle, ease of use of fuzzers should be improved so that they are easy to integrate into continuous integration pipelines. This would allow catching the bugs earlier in the software development process.

New injection methods for fuzzers is another important topic for future work. Currently there are many wired and wireless protocols that simply cannot be fuzz tested because there is no way to reliably inject fuzzed packets into the target system.

6. SUMMARY

Bluetooth LE is nowadays increasingly common technology within IoT devices and becoming even more mainstream with upcoming LE Audio. During last few years increasing amount of vulnerabilities have been identified in lowest layers of Bluetooth but still there have not been commercial grade fuzzing solution for Bluetooth LE link layer.

Objective of this thesis was to design and implement a solution which allows injection of Bluetooth LE LL packets over the air to target device to enable fuzzing of Bluetooth LE link layer implementations. Two different approaches for Bluetooth LE LL injection were designed and implemented. Both approaches were built using Zephyr OS as a base and by modifying the Bluetooth LE LL stack. Both approaches utilised vendor specific HCI commands and events for providing an API for LL packet injection to host stack. First approach was based on an idea that LL implementation stays fully in controller and arbitrary LL packet can be injected by replacing the valid packet with anomalised one. Second approach instead moved the responsibility of LL state handling up to host stack and controller just blindly acted as a proxy passing incoming LL packets to host stack and LL packets coming from host stack over the air to the remote device.

As a main outcome of this thesis, firmware for a solution was designed and implemented. The solution utilises combination of both of the approaches presented earlier. In addition different hardware options were studied and after careful evaluation Laird BL654 USB dongle was selected. The combination of hardware and firmware will be used as a essential part of commercial fuzz testing product, Synopsys Defensics.

The developed solution was evaluated against state of the art of the industry: Greyhound and InternalBlue security evaluation frameworks. The results prove that the developed solution is on-par with state of the art on this field.

7. REFERENCES

[1] Classen J. & Hollick M. (2019) Inside job: Diagnosing bluetooth lower layers using off-the-shelf devices. CoRR abs/1905.00634. URL: http://arxiv.org/abs/1905.00634.

[2] (2020), Bluetooth market update 2020. URL: https://www.bluetooth.com/bluetooth-resources/2020-bmu/, Bluetooth Special Interest Group. Accessed 23.1.2021.

[3] Myers G.J., Sandler C. & Badgett T. (2011) The Art of Software Testing. Wiley Publishing, 3rd ed.

[4] Takanen Ari a. (ed.) (2018) Fuzzing for software security testing and quality assurance. Artech House information security and privacy series, Artech House, Boston, Massachusetts; London, England, second edition ed., 318 p. Includes index.

[5] (2019), Bluetooth core specification version 5.2. URL: https://www.bluetooth.com/specifications/bluetooth-core-specification/, Bluetooth Special Interest Group. Accessed 23.1.2021.

[6] Kindt P., Yunge D., Diemer R. & Chakraborty S. (2014) Precise energy modeling for the bluetooth low energy protocol. ACM Transactions on Embedded Computing Systems 19.

[7] Bluez. URL: http://www.bluez.org/, BlueZ Project. Accessed 23.1.2021.

[8] Apache mynewt nimble. URL: https://mynewt.apache.org/latest/network/, Apache Foundation. Accessed 23.1.2021.

[9] Btstack. URL: https://github.com/bluekitchen/btstack/, BlueKitchen BTstack. Accessed 23.1.2021.

[10] Packetcraft protocol software. URL: https://github.com/packetcraft-inc/stacks, Packetcraft Protocol Software. Accessed 23.1.2021.

[11] Gupta N.K. (2016) Inside Bluetooth Low Energy, Second Edition., vol. 2nd ed. Artech House. URL: http://pc124152.oulu.fi:8080/login?url=.

[12] Android developer documentation. URL: https://developer.android.com/reference/android/bluetooth/package-summary, Google LLC. Accessed 26.4.2021.

[13] ios developer documentation. URL: https://developer.apple.com/documentation/corebluetooth, Apple Inc. Accessed 26.4.2021.

[14] (1990) Ieee standard glossary of software engineering terminology. IEEE Std 610.12-1990 , pp. 1–84.

[15] Singh Y. (2011) Software Testing. Cambridge University Press. URL: http://pc124152.oulu.fi:8080/login?url=.

[16] Miller B., Fredriksen L. & So B. (1990) An empirical study of the reliability of unix utilities. Commun. ACM 33, pp. 32–44.

[17] Kaksonen R. (2001) A functional method for assessing protocol. Implementation security: Licentiate thesis. Ph.D. thesis, Aalto University, Finland.

[18] Miller B.P., Zhang M. & Heymann E.R. (2020) The relevance of classic fuzz testing: Have we solved this one? CoRR abs/2008.06537. URL: https://arxiv.org/abs/2008.06537.

[19] Synopsys defensics. URL: https://www.synopsys.com/software-integrity/security-testing/fuzz-testing.html, Synopsys Inc. Accessed 23.1.2021.

[20] (2014), Heartbleed website. Website. URL: https://heartbleed.com/. Accessed 1.4.2021.

[21] Project ubertooth. URL: https://github.com/greatscottgadgets/ubertooth. Accessed 26.1.2021.

[22] nrf sniffer. URL: https://www.nordicsemi.com/Software-and-tools/Development-Tools, Nordic Semiconductor. Accessed 6.4.2021.

[23] Ryan M. (2013) Bluetooth: With low energy comes low security. In: 7th USENIX Workshop on Offensive Technologies (WOOT 13), USENIX Association, Washington, D.C. URL: https://www.usenix.org/conference/woot13/workshop-program/presentation/ryan.

[24] Cyr B., Horn W., Miao D. & Specter M. (2014) Security analysis of wearable fitness devices (fitbit). Massachusetts Institute of Technology 1.

[25] Cope P., Campbell J. & Hayajneh T. (2017) An investigation of bluetooth security vulnerabilities. In: 2017 IEEE 7th Annual Computing and Communication Workshop and Conference (CCWC), pp. 1–7.

[26] Mantz D. (2018) InternalBlue - A Bluetooth Experimentation Framework Based on Mobile Device Reverse Engineering. Master's thesis. URL: http://tubiblio.ulb.tu-darmstadt.de/107125/.

[27] Mantz D., Classen J., Schulz M. & Hollick M. (2019) Internalblue - bluetooth binary patching and experimentation framework. CoRR abs/1905.00631. URL: http://arxiv.org/abs/1905.00631.

[28] Garbelini M.E., Wang C. & Chattopadhyay S. (2020) Greyhound: Directed greybox wi-fi fuzzing. IEEE Transactions on Dependable and Secure Computing (Early Access). DOI: http://dx.doi.org/10.1109/TDSC.2020.3014624.

[29] Garbelini M.E., Wang C., Chattopadhyay S., Sun S. & Kurniawan E. (2020) Sweyntooth: Unleashing mayhem over bluetooth low energy. In: USENIX Annual Technical Conference (USENIX ATC). URL: `https://asset-group.github.io/papers/SweynTooth.pdf`.

[30] Abbasi A., Wetzels J., Holz T. & Etalle S. (2019) Challenges in designing exploit mitigations for deeply embedded systems. In: 2019 IEEE European Symposium on Security and Privacy (EuroS P), pp. 31–46. DOI: `https://doi.org/10.1109/EuroSP.2019.00013`.

[31] Antonioli D., Tippenhauer N. & Rasmussen K. (2019), Knob attack. Website. URL: `https://knobattack.com/`. Accessed 20.4.2021.

[32] Antonioli D., Tippenhauer N.O. & Rasmussen K.B. (2019) The KNOB is broken: Exploiting low entropy in the encryption key negotiation of bluetooth br/edr. In: 28th USENIX Security Symposium (USENIX Security 19), USENIX Association, Santa Clara, CA, pp. 1047–1061. URL: `https://www.usenix.org/conference/usenixsecurity19/presentation/antonioli`.

[33] (2018), Internalblue github repository. Website. URL: `https://github.com/seemoo-lab/internalblue/`. Accessed 20.4.2021.

[34] Wireshark. URL: `https://www.wireshark.org/`, Wireshark Foundation. Accessed 19.5.2021.

[35] Zephyr project. URL: `https://www.zephyrproject.org/`, Zephyr Project. Accessed 23.1.2021.

[36] Zephyr technical documentation. URL: `https://docs.zephyrproject.org/latest/introduction/`, Zephyr Project. Accessed 23.1.2021.

[37] (2014), Todo website. Website. URL: `https://heartbleed.com/`. Accessed 1.4.2021.

[38] Zephyr github repository. URL: `https://github.com/zephyrproject-rtos/zephyr`, Zephyr Project. Accessed 23.1.2021.