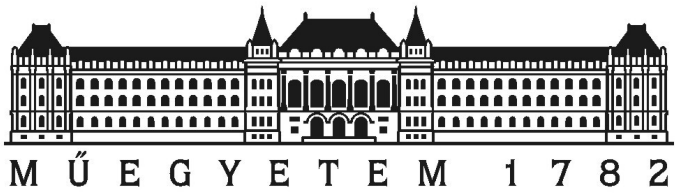

Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar
Szélessávú Hírközlés és Villamosságtan Tanszék

Inspecting the robustness of BLE protocol with fuzz testing

Author: Kepics János
Advisor: Dr. Csilling Ákos (Robert Bosch Kft.),
Dr. Horváth Bálint (HVT)

2022

Contents

1 The need for cybersecurity in wireless protocols **5**
1.1 Cybersecurity attacks in the wild . . . 5
1.1.1 Nest Thermostat . . . 5
1.1.2 Philips Hue . . . 5
1.1.3 Insulin Pump . . . 6
1.1.4 Smart Door Locks . . . 6
1.1.5 Tesla . . . 7
1.1.6 The Jeep hack . . . 7
1.2 Reasons and approaches to vulnerabilities . . . 8

2 Embedded security evaluation **9**
2.1 Theoretical security analysis . . . 10
2.1.1 Design analysis . . . 10
2.1.2 Threat and risk analysis . . . 10
2.2 Practical security testing . . . 10
2.2.1 Functional security testing . . . 10
2.2.2 Penetration testing . . . 11

3 Fuzz testing **12**
3.1 Structure . . . 12
3.1.1 Message generator . . . 12
3.1.2 Message publisher . . . 13
3.1.3 Target monitor . . . 13
3.2 Taxonomy of fuzzing . . . 13
3.2.1 Based on prior knowledge . . . 14
3.2.2 Based on fuzzed data generation . . . 14
3.2.3 Based on data generation intelligence . . . 15
3.3 Fuzzing embedded systems . . . 16

4 Bluetooth Low Energy **17**
4.1 Introduction . . . 17
4.2 Host . . . 18
4.3 Controller . . . 19
4.3.1 Physical layer . . . 19
4.3.2 Link layer . . . 19

5 Design and implementation **23**
5.1 Motivation and specification . . . 23
5.2 State of the art . . . 24
5.3 Implementation . . . 25

5.3.1 Message generator 25
5.3.2 Message publisher 27
5.3.3 Target monitor 28

6 Results of the execution **30**

7 Conclusion and further development **33**
7.1 Conclusion 33
7.2 Future development 33
7.2.1 Side-channel analysis during fuzzing 33
7.2.2 Handle statefulness 33

Kivonat

Mindennapjainkban rengeteg Internet of Things (IoT) eszközzel találkozunk, beleértve ebbe a különbféle fitness trackereket, okosgyűrűket, -órákat. De nem csak fogyasztói termékek, hanem sok egészségügyi eszköz és autóipari termék is ebbe a kategóriába tartozik, mint például pacemakerek, telefonos kulcs applikációk és vezeteték nélküli kommunikációt használó szenzorok.
A legtöbb rövid hatótávolságú, kisfogyasztású IoT eszköz a BLE (Bluetooth Low Energy) protokollt használja a központi eszközzel való kommunikációhoz. Ezen a kapcsolaton belül a legtöbb esetben nagyon személyes információkat oszt meg a készülék a felhasználóról.
A BLE protokollnak sok sérülékeny pontját megtalálták már korábbi kutatások fuzz teszteléssel.
Fuzzing (vagy fuzz tesztelés) az egyik legerőteljesebb automatizált szoftver tesztelési eljárás.A módszer lényege, hogy a tesztelés alatt álló eszköznek véletlenszerűen módosított, de helyes üzeneteket küldünk, majd ezek hatását különböző eljárásokkal vizsgáljuk. A teszt célja egy nem várt viselkedés észlelése. A módszer nagy előnye, hogy a vizsgálandó rendszer pontos ismerete nélkül is alkalmazható, mert a kimenet helyességét nem kell ellenőrizni.
A TDK dolgozatom keretein belül egy Bluetooth Low Energy okos fuzzer-t mutatok be. Dolgozatomban elsőként definiálom a szükséges elméleti hátteret: A különböző típusú fuzz teszter eszközöket és a BLE protokoll struktúráját. A kiberbiztonsági támadásokat vezetéknélküli kommunikáció során és a biztonsági tesztelést, mint ezeknek a megelőző lépését. A bemutatás során részletesen ismertetem az üzenet generátort, amely felépíti a mutálódott üzeneteket, az üzenet publikálót és a céleszköz figyelőt, amelyet a tesztelt eszköz ébren tartásához és állapotának ellenőrzésére implementáltam.
Végül, de nem utolsó sorban bemutatom az egyes BLE specifikus teszt tipusokat amelyekkel a protokoll sérülékenységét ellenőrzöm és az elért eredményeket, amelyet IoT eszközök tesztelése során kaptam.

Abstract

Internet of Things (IoT) devices manifest in a variety of forms today, including fitness trackers, rings, smart watches, but it is not just in the consumer industry. There are medical-, such as pacemakers and automotive IoT devices, like sensors and smart keys.
Most short range, low power IoT devices use the BLE (Bluetooth Low Energy) protocol to communicate with a master device. This communication link can contain very personal information about the user. Several vulnerabilities exist for BLE and most of them were found with fuzz testing by researchers.
Fuzzing (or fuzz testing) is considered as one of the most powerful automated software testing methods. This technique relies on sending malformed messages to the target application in order to provoke unexpected behaviour or failures. The simplicity of the method comes from the fact that the output is not checked, therefore it can be applied without a detailed understanding of the device under test.
In this TDK paper I present a Bluetooth Low Energy smart fuzzer, developed to test robustness and security.
First, I define the needed theoretical background for this paper: The different type of fuzzing tools and the structure of Bluetooth Low Energy protocol. Relevant cybersecurity attacks for wireless communication protocols, and security testing to prevent them.
Second, I present the detailed functionality of my fuzzer with focus on the message generator, which prepares the mutated message, the message publisher and the target monitor, implemented to keep alive the connection and to give feedback to the systems.
Finally, I show BLE specific test cases which checks for vulnerability in different areas of the protocol and the results that were acquired during testing the tool on several IoT devices.

Chapter 1

The need for cybersecurity in wireless protocols

To discover cybersecurity and why wireless device developers should pay attention to it I am going to look at previous real-life examples where cybersecurity vulnerabilities were exploited. After this chapter the reader should have a general understanding on what type of attacks and security issues should be expected in a product.

1.1 Cybersecurity attacks in the wild

1.1.1 Nest Thermostat

The Nest Thermostat is a smart device designed to control a central air conditioning unit, based on heuristic and learned behaviour. Coupled with a Wi-Fi module, the Nest Thermostat can connect to the user's home or office network and interface with the Nest Cloud, thereby allowing for remote control of the unit. It also includes a ZigBee module for communication with other Nest devices.[1]
Jason Doyle, security researcher, identified a vulnerability in Nest products that involved sending a custom-crafted value in the Wi-Fi service set identifier (SSID) details via Bluetooth to the target device, which would then crash the device and ultimately reboot it. This would also allow a burglar to break into the house during the duration of the device reboot (around 90 seconds) without being caught by the Nest security system.[2]

1.1.2 Philips Hue

Philips hue is a personal wireless system which contains wireless LED light bulbs and a wireless bridge.The sytem can be configured to any of the 16 million colours and any Android or iOS user can use it.
In August 2013, Nitesh Dhanjani, a security researcher, made a security evaluation on the Philips hue[3], in which he came up with a novel technique to cause a permanent blackout by using a replay attack. He discovered this vulnerability after he realized that the Philips Hue smart devices were only considering the MD5 of the media access control (MAC) address as the single parameter to validate the authenticity of a message. Since the attacker can very easily learn the MAC address of the legitimate host, he or she can craft a malicious packet indicating that it came from the genuine host and with the command to turn the bulb off. Doing this continuously would allow the attacker to cause a permanent blackout with the user having no other option than to replace the light bulb.

Philips Hue uses a technology called ZigBee to exchange data between the devices while keeping resource consumption to a minimum. The same attack that was possible on the device using Wi-Fi packets would also be applicable to ZigBee. In the case of ZigBee, all an attacker would need to do is simply capture the ZigBee packets for a legitimate request, and simply replay it to perform the same action at a later point in time and take over the device.[4]

1.1.3 Insulin Pump

A security researcher named Jay Radcliffe identified [5] that some medical devices, specifically insulin pumps, could be suffering from a replay-based attack vulnerability. Radcliffe, a Type 1 diabetic himself, set out to research one of the most popular insulin pumps on the market, the OneTouch Ping insulin pump system by Animas, a subsidiary of Johnson Johnson. During the analysis, he found that the insulin pump used cleartext messages to communicate, which made it extremely simple for anyone to capture the communication, modify the dosage of insulin to be delivered, and retransmit the packet. When he tried out the attack on the OneTouch Ping insulin pump, it worked flawlessly, with there being no way of knowing the amount of insulin that was being delivered during the attack.[4]

1.1.4 Smart Door Locks

Security researchers Anthony Rose and Ben Ramsey of the security firm Merculite gave a presentation titled "Picking Bluetooth Low Energy Locks from a Quarter Mile Away" at DEF CON 24 [6] in which they disclosed vulnerabilities in several smart door lock products, including Quicklock Padlock, iBluLock Padlock, Plantraco Phantomlock, Ceomate Bluetooth Smart Doorlock, Elecycle EL797 and EL797G Smart Padlock, Vians Bluetooth Smart DoorlockOkidokey Smart Doorlock, Poly-Control Danalock Doorlock, Mesh Motion Bitlock Padlock, and Lagute Sciener Smart Doorlock.
The vulnerabilities discovered by Rose and Ramsey were of varying types including transmission of the password in clear text, susceptibility to replay-based attacks, reversing mobile applications to identify sensitive information, fuzzing, and device spoofing. For instance, during the process of resetting the password, Quicklock Padlock sends a Bluetooth low energy (BLE) packet containing the opcode, old password, and new password. Because even normal authentication happens over clear text communication, an attacker can then use the password to set up a new password for the door lock that would render the device useless for the original owner. The only way to reset it would be to remove the device's battery after opening the enclosure. In another device, the Danalock Doorlock, one can reverse engineer the mobile application to identify the encryption method and find the hard-coded encryption key ("thisisthesecret") being used. [4]

Figure 1.1: From left to right: Nest Thermostat[7], Philis Hue[8], OneTouch Ping[9], OK-LOK Smart Lock[10]

1.1.5 Tesla

Tesla Model 3 and Model Y employ a Bluetooth Low Energy (BLE) based passive entry system. This system allows users with an authorized mobile device or key fob within a short range of the vehicle to unlock and operate the vehicle, with no user interaction required on the mobile device or key fob. This system infers proximity of the mobile device or key fob based on signal strength (RSSI) and latency measurements of cryptographic challenge-response operations conducted over BLE.

NCC Group has developed a tool for conducting a new type of BLE relay attack operating at the link layer, for which added latency is within the range of normal GATT response timing variation, and which is capable of relaying encrypted link layer communications. This approach can circumvent the existing relay attack mitigations of latency bounding or link layer encryption, and bypass localization defences commonly used against relay attacks that use signal amplification. As the latency added by this relay attack is within the bounds accepted by the Model 3 (and likely Model Y) passive entry system, it can be used to unlock and drive these vehicles while the authorized mobile device or key fob is out of range.[11]

1.1.6 The Jeep hack

Last, but not least I have to reference probably one of the most popular IoT and automotive hack of all time: The Jeep Hack.
Two security researchers, Dr. Charlie Miller and Chris Valasek, demonstrated in 2015 how they could remotely take over and control a Jeep using vulnerabilities in Chrysler's Uconnect system, resulting in Chrysler having to recall 1.4 million vehicles.[13]
The complete hack took advantage of many different vulnerabilities, including extensive efforts in reverse engineering various individual binaries and protocols. One of the first vulnerabilities that made the attack possible was the Uconnect software, which allowed anyone to remotely connect to it over a cellular connection. NavTrailService was found to have an execute method that allowed the researchers to run arbitrary code on the device. Once arbitrary command execution was gained, it was possible to perform a lateral movement and send CAN messages taking control of the various elements of the vehicle, such as the steering wheel, brakes, headlights, and so on.[4]

1.2 Reasons and approaches to vulnerabilities

As you can see there are several protocols, which offer similar services and frameworks. IoT is a huge, growing industry on its own, but also by its nature it is connected to other industries like the medical- and the automotive field, therefore a lot of companies would like a piece from the IoT pie. These protocols include: Bluetooth, Bluetooth Low Energy, Wi-Fi, ZigBee, ZWave, LoRa etc. There is not one company which has monopoly over the market, thus there is competition, which drives innovation. However, most of these protocols are not mature enough to have good security.
From my search for different cybersecurity attacks I mostly encountered these four methods:

- Eavesdropping: As the name suggests, eavesdropping refers to a third party device listening in on the data that's being exchanged between two paired devices.[14] To an extent every attack includes an eavesdropping phase.

- Man in the Middle/Relay/Replay Attacks: Man in the middle attacks involve a third party device impersonating a legitimate device, tricking two legitimate devices into believing that they're connected to each other, when in reality, the legitimate devices are connected to the impersonator. This sort of an attack enables the attacker/impersonator to access all the data that is being exchanged between the devices and, also to manipulate the data by deleting it or changing it, before it reaches the respective device.[14](such as 1.1.2, 1.1.3, 1.1.5)

- Denial of Service: Since most wireless devices these days work on inbuilt battery packs, these devices run the risk of being exposed to Denial of Service Attacks (DoS). DoS attacks expose a system to the possibility of frequent crashes leading to a complete exhaustion of its battery.[14] (such as 1.1.1, 1.1.2)

Chapter 2

Embedded security evaluation

As discussed in Chapter 1, modern devices can be open to various security risks. By applying in-depth security evaluation, potential security weaknesses can be identified and countered before an attacker can exploit this weakness in the field and cause real financial or even personal damages. The earlier such a security evaluation is done within the development cycle, the less costly and time-consuming it is to identify and close security weaknesses. 2.1. figure from [15] shows a tree diagram on the categories of the possible techniques. Security evaluation can be done in theory as well as in practice (this can be segmented even further).

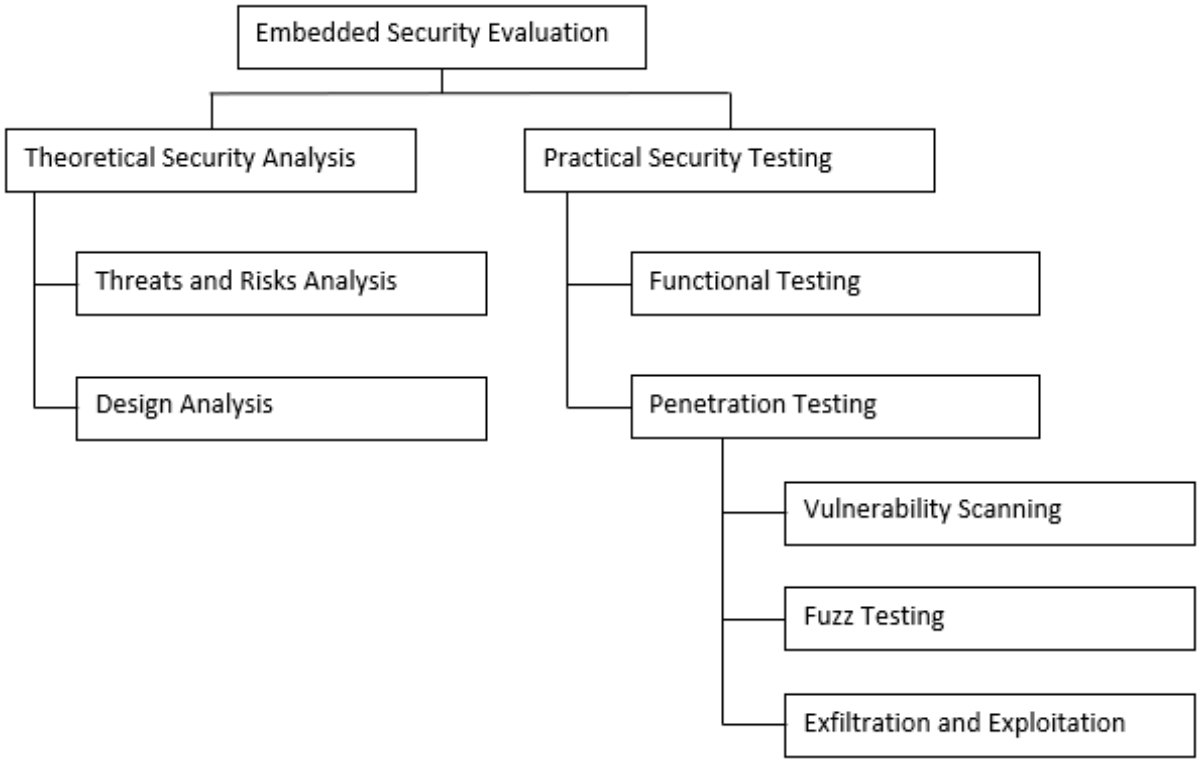

Figure 2.1: Overview of embedded security evaluation

2.1 Theoretical security analysis

Theoretical security evaluation can (and should) be done during virtually all steps of the development cycle. Depending on the level of scrutiny and the documents available, we differentiate between a more high-level design analysis and an in-depth threat and risk analysis. If the documentation is insufficient or contains third-party components, then the risk assessment won't give the full picture. Theoretical security evaluation fails to find implementation flaws or deviations from the specification in the implementation.

2.1.1 Design analysis

- Goals: Establish the soundness of the system architecture, find out systematic flaws. Inspecting potential attack vectors like weak cryptography algorithms or bad interactions of different protocols.
- Advantage: Only theoretical description of the system is needed.
- Disadvantage: The depth of the analysis varies on the provided system level.

2.1.2 Threat and risk analysis

- Goal: Categorise the identified vulnerabilities further and prioritise detected flaws according to their impact on the system.
- Steps: Possible attack vectors are identified, the difficulty of this attack is rated based on required time, needed expertise, equipment and necessary access level. Evaluate potential damage of the attack.

2.2 Practical security testing

Practical security testing, of course, can only be conducted on an implementation of the target system, for instance with a first prototype. Practical security testing contrary to theoretical security analysis can detect implementation errors that could be exploited by an outside attacker but also unspecified functionality and deviations from the specifications. Therefore, a thorough practical security test helps to establish trust in the soundness of the implementation. Furthermore, this type of test helps to estimate the actual difficulty of an attack against the target system.
However, practical security testing cannot give any claim on completeness against security attacks. Depending on the time and resources it is possible to miss larger systematic flaws.

2.2.1 Functional security testing

- Tests all security-related functions inside the test system for correct behaviour and robustness. Similar to general functional testing but with focus on security functionality.
- Goals: can find implementation errors, deviations from the specification, but especially unspecified functionality can't be tested. All of these errors might result in a potential security weakness.

2.2.2 Penetration testing

Consists of at least three steps:

- Vulnerability scanning: tests the system for already known common security vulnerabilities, for instance, known security exploits or configurations with known weaknesses.
- Fuzz testing: Tries to find new vulnerabilities of an implementation by sending systematically malformed input to the target system to check for unknown, potentially security-critical system behaviour.
- Exfiltration and exploitation: To test the security of the whole system. During this step, a "smart human tester", i.e. a skilled individual with profound expertise in the areas of IT security, electronic engineering, computer science, and automotive systems, tries to exploit all the vulnerabilities which were identified in the earlier steps in a sophisticated way based on many years of "hacking experience" with the goal to change the behaviour of the target system.

Chapter 3

Fuzz testing

Fuzzing or, fuzz-testing, is the method that finds vulnerabilities and bugs by inserting randomly crafted inputs into a target, named SUT (System Under Test). These specially crafted inputs trigger unexpected behaviour in the SUT, and let us find bugs, such as faulty memory violations, assertion violations, incorrect null handling, deadlocks, infinite loops, undefined behaviours, or incorrect managements of other resources.[17]
Fuzzing is the most efficient security testing method as it can be semi- and fully automated, therefore it can be performed at large scale and unattended.

3.1 Structure

To achieve an modular system the process of fuzzing should be divided into three stages[15], depending on the complexity of our fuzzing strategy there can be more phases.

Figure 3.1: Structure of fuzz testing

3.1.1 Message generator

In this stage the fuzzer creates invalid messages which will be sent to the target. This is the most critical stage of fuzzing: the performance of our search therefore the quality of our fuzzing correlates to a great extent with the message generator. However, this stage stands mostly independent of the system under test (SUT) consequently it can be placed into other fuzzers with relative ease.
First a pseudo-random seed is created, then the fuzzer uses the seed to create the fuzzed data, which gets sent out. Afterwards, the process is repeated.

3.1.2 Message publisher

The goal of this stage is to send the previously generated data to the SUT by using the SUT specific media. In contrast to the message generator, the message publisher is developed specially for that media (protocol), but its function is the easiest to define: send out messages.

3.1.3 Target monitor

In this phase the behaviour of the SUT is monitored in order to detect unexpected outputs or crashes that could be related to triggered vulnerabilities. This involves feedback from the target upon which to base the judgement whether the SUT was able to cope with the input.
Possible feedback could be to send requests to the target to check it's responsiveness, send valid input after the malicious ones to test for correct behaviour etc.
Once a bad response is found, it is the target monitors job to narrow down the possible inputs which could cause it. Checking for fault is also a function of this stage.
Advanced functionalities can make the fuzzing quality better. For example it can involve providing feedback for the message generator and tracing the progress in the test code coverage (if provided).

3.2 Taxonomy of fuzzing

The classification [17] should be done according to each of the predefined stages and should answer the following questions:

- How much prior knowledge does the stage have of the SUT?
- What is the fuzzed data generation approach?
- What is the extent for the data generation intelligence?

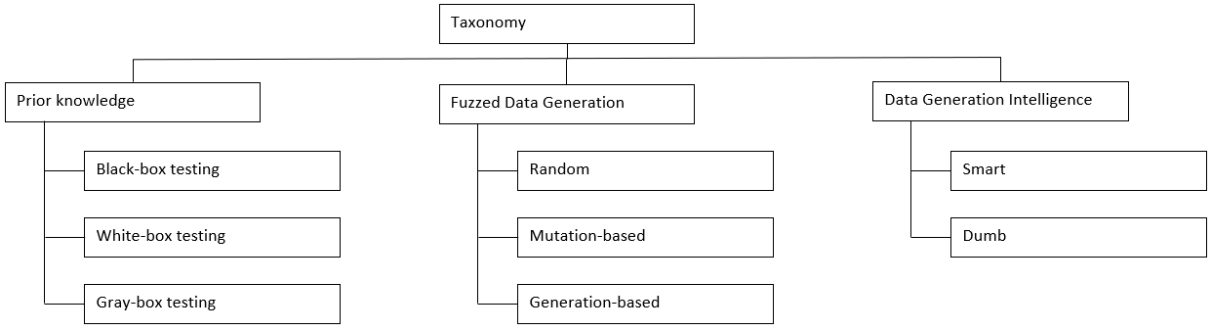

Figure 3.2: Taxonomy of fuzzing

3.2.1 Based on prior knowledge

This criteria considers the prior knowledge available regarding the SUT, based on this, the fuzzing techniques can be divided into three main categories.

Black-box testing

The fuzzer has no prior knowledge of the SUT. (It hasn't got access to the source code or the internal logic.)
The main advantage of black-box testing is that it can be applied against virtually any target. However, the performance of black-box testing is low and usually it can't find complex vulnerabilities.

White-box testing

Contrary to black-box testing in this case source code and the internal knowledge is known by the fuzzer for every part of the SUT.
The main advantage is that it provides complete code coverage. Though as good as it sounds this approach is much more complex than black-box testing, therefore developing this method is much harder. In addition, code access is required, which is not available in most products as they usually contain third-party software.

Gray-box

This solution offers the best of both words as in gray-box testing has partial access to the internal logic of the SUT. Its execution logic is similar to that of black-box fuzzers, but it can leverage some limited information (often gathered through measurements) about the SUT to improve fuzzing performance and coverage.
Nowadays, gray-box testing is the most widespread method, as it does not require full source-code access but it is able to infer information that makes it more efficient than black-box testing.

3.2.2 Based on fuzzed data generation

As I already said the message generator phase is the most crucial phase in determining fuzzing performance. Based on the fuzzed data generation approach the fuzzer can be categorised.

Random

This is the base standard for every fuzzer. The system does not take formatting the random data in any shape or form to the target's needs into consideration, it generates random data of random length. This is the easiest setup in challenging input validation but restricted in terms of its potential to penetrate protocols with complex message structure.

Mutation-based

This approach generates the new inputs from the previously generated test cases.
The first time an initial seed is created, which will be slightly altered recursively. One alteration is called a mutation. It is a key decision to decide how to perform the mutation, which will influence the performance of fuzzing. If a blind approach is chosen with no feedback the mutation won't take advantage of the previous states of the target which could be used to steer the mutation. There are different strategies for input mutation and improving mutation performance is an active research field [18], [19], [20], [21].

For this generation strategy, it is not necessary to know the specifications of the input data or protocol. It is still relatively easy to setup and can provide more efficiency than full randomness, but many messages will fail for protocols with checksum or other hard checks.

Generation-based

As a first look this approach seems the same as mutation-based fuzzing, because it is a subclass of it, where the message generator uses a set of specifications on the SUT inputs (the message generator output) to generate new fuzzed test data. In contrast to the mutation-based generation, it is necessary to know the syntax of the SUT inputs, including their format and used protocol. so, a first phase called preparation gets introduced to the fuzzing structure. This approach takes advantage in having access to the specification of the tested protocol and/or valid data collection is possible from the target.
The function of the message fields is known which provide a more in depth look at the protocol, thus more complex protocols can be implemented with dependencies such as check sums supplying an outwardly verified message.
However, with complexity come challenges, consequently implementing generation-based fuzzing is much harder than its counterpart. The preparation phase can be very time consuming, automation is hard to achieve.

3.2.3 Based on data generation intelligence

The intelligence is related to its ability to generate new input data taking into account the feedback it receives after an execution of a test. The feedback helps to improve the new input generation, as it can be used to decide which part of the test case should be modified, and how to modify it. Based on the previous statement the intelligence of the fuzzer is determined by the target monitor.

Smart fuzzers

Smart fuzzers consider the behaviour of the SUT during the fuzzing process. By first defining a correct behaviour then inspecting how the fuzzed data affected this behaviour (e.g. crashes) and taking the gathered information into account during the next message generation.
In general smart fuzzers are more efficient in detecting vulnerabilities.

Dumb fuzzers

Dumb fuzzers do not consider feedback from the previous executions as input for new data generation. As they do not have to receive feedback, analyse it, and act based on it, dumb fuzzers are faster test executors than their smart counterparts, but less effective when considering vulnerability search.

3.3 Fuzzing embedded systems

One common thing between the devices mentioned in Chapter 1 is that they are embedded systems.
Embedded system fuzzing[16] carries it's own unique challenges. It tends to happen that vulnerabilities discovered by fuzzing will be caused by implementation flaws in the firmware as most of the time developers use open-source components in firmware development without any update timeline. The nature of this development sacrifices security to launch the product as soon as possible. To accentuate this problem, patching vulnerabilities after product launch is a hard process which can involve recalling of hundreds of products in the automotive field. The IoT field usually doesn't carry out recalling of products so if the IoT device is not connected to a reliable network then remote patching is impossible, it is hence crucial to discover such vulnerabilities in the development cycle and fix them before an attacker does.
To design an effective and efficient fuzzing method for embedded systems, several challenges must be overcome.

- Lack of a feedback mechanism: Without access to firmware, it is nearly impossible to obtain the internal execution of functions. Consequently, we need creative solutions to obtain feedback from a device to optimize the data generation process.
- Diverse message formats: There is not a general protocol which is used, so the solution should be able to infer the format from a raw message or the specific protocol must be implemented in the message generator.
- Randomness in responses: The response messages may contain random elements, such as timestamps or tokens. Such randomness results in different responses for the same message, which could diminish the effectiveness of fuzzing if these fields are not interpreted correctly during the message generation.

Chapter 4

Bluetooth Low Energy

4.1 Introduction

For this project I worked on implementing fuzzing the Bluetooth Low Energy protocol so in this chapter I would like to give an overview on the protocols usage and the stack(4.1). BLE has a more than 400 page long documentation[22] so I will try to highlight project specific information.
Under the umbrella of Bluetooth we can actually refer to two different protocols: Bluetooth Classic and Bluetooth Low Energy (a.k.a. BLE). Table 4.1 shows the differences between Classic and BLE. There is a significant difference in application throughput: BLE is only capable of 0.27 Mbps whereases classic can achieve 2.1 Mbps. However, BLE communication consumes much less power (0.1-0.5W) than Classic (1W), thus IoT devices use this over Classic as they are powered by a coin cell battery most of the time.

Specification	Bluetooth Classic	Bluetooth Low Energy (BLE)
Range	100 m	Greater than 100 m
Data Rate	1-3 Mbps	1-2 Mbps
Application Throughput	0.7-2.1 Mbps	0.27 Mbps
Frequency	2.4 GHz	2.4 GHz
Robustness	Adaptive fast frequency hopping, FEC, fast ASK	24-bit CRC, 32-bit Message Integrity Check
Latency	100 ms	6 ms
Time lag	100 ms	3 ms
Voice capable	Yes	No
Network Topology	Star	Star
Power Consumption	1 W	0.01-0.5 W

Table 4.1: Bluetooth Classic and BLE

4.1. figure shows the BLE protocol stack. The stack can be separated into three segments application layer, host and controller. A BLE application programmer mainly interacts with the application layer and some of the Host layers to define functions to the service they provide. IC manufacturers implement the link layer and physical layer that is why it is called the controller section.

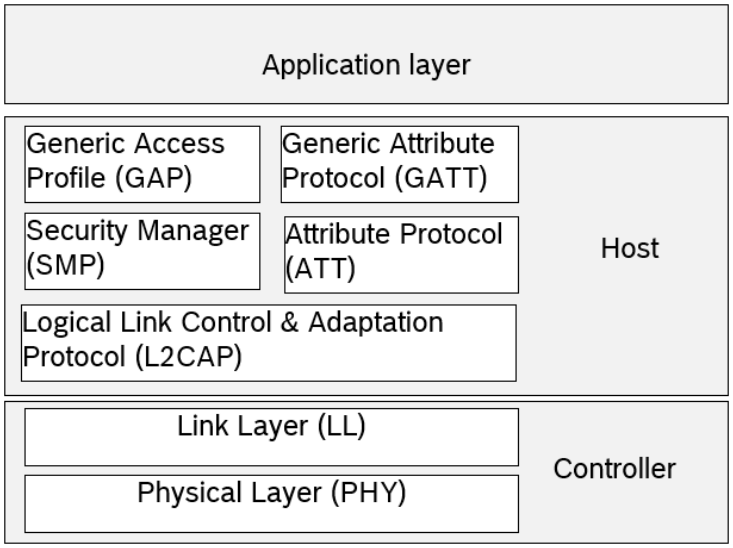

Figure 4.1: BLE stack

Most of the project is done on the lower layers of the protocol (especially link layer), but to provide a more complete outlook on the BLE protocol I would like to give a brief overview on the other layers as well before deep diving into the controller section.

4.2 Host

- **HCI**: It provides communication between controller and host through standard interface types. This HCI layer can be implemented either using API or by interfaces such as UART/SPI/USB. Standard HCI commands and events are defined in the bluetooth specifications. [23]
- **ATT**: This layer allows BLE device to expose certain pieces of data or attributes.[23]
- **GATT**: This layer is service framework which specifies sub-procedures to use ATT. Data communications between two BLE devices are handled through these sub-procedures. The applications and/or profiles will use GATT directly. [23]
- **GAP**: This layer directly interfaces with application layer and/or profiles on it. It handles device discovery and connection related services for BLE device. It also takes care of initiation of security features. [23]
- **SMP**: This layer provides methods for device pairing and key distributions. It offers services to other protocol stack layers in order to securely connect and exchange data between BLE devices. [23]
- **L2CAP**: This layer offers data encapsulation services to upper layers. This allows logical end to end data communication.[23]

4.3 Controller

4.3.1 Physical layer

BLE operates in the ISM (Industrial Scientific Medical) band, uses FSK modulation and frequency hopping.

$$f_{carrier} = 2402 + k \cdot 2MHz \text{ where } k = 0....39$$

In FSK modulation the information correlates with the deviation from the carrier frequency. Binary 1 represented by a positive-, 0 by negative frequency deviation. If the data rate is 1Mbps the deviation should be higher then 185kHz, for 2Mbps 370kHz.

Frequency hopping means that after ever package a new carrier frequency is chosen based on a predefined algorithm. The used channels collection is called channel map.

4.3.2 Link layer

The link layer sits directly above the physical layer, responsible for advertising, scanning, and creating/maintaining connections.

Link layer state graph

The best way to define link layer responsibilities is by a state graph (4.2). A device can't operate in different states simultaneously.
The description of states:

- **Stand by**: In this state messages can't be transmitted or received by the device. It can be entered by any state.
- **Advertising**: In this state the pheripheral device sends out advertisement packets to be noticed by other devices, thus the device in this state is called advertiser.
- **Scanning**: In this state the devices searches for packets which are sent by the advertiser, thus the device in this state is called scanner.
- **Initiator**: In this state the device listens for advertisement packets from predefined devices, thus the device is called an initiator.
- **Synchronization**: In this state the device listens for periodic specialised advertisement packets from a predefined device.
- **Connection**: This state can only be entered by an initiator or an advertiser. To get into this state the initiator must respond to the advertiser with a connection request.
  From that point onwards:
  - The initiator becomes the master.
  - The advertiser becomes the slave.

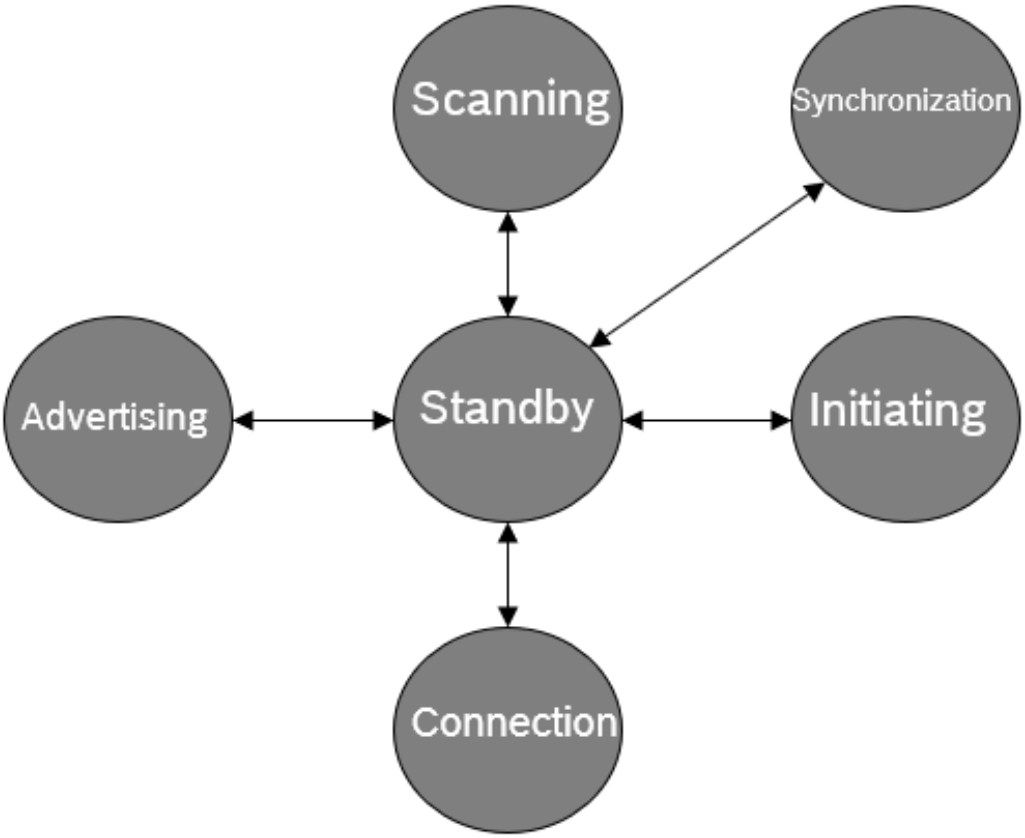

Figure 4.2: Link layer state graph

Link layer messages exchange

Based on the previous accounts the build-up of a connection in link layer is stateful and deterministic to an end.
Figure 4.3 shows this process in a block diagram. The precondition of this diagram is that the central(C) device would like to connect to the peripheral(P) device, so first it becomes a scanner and ideally finds the advertisement packets from the peripheral acting as an advertiser. Once the scanner finds the advertiser it goes into initiator state and sends out a scan request directly to the found advertiser, which will respond with a scan response. After receiving the scan response, the initiator sends a connection request to the advertiser, thus the peripheral becomes the slave and the central the master in the newly created connection. Finally, the last makeable steps staying only in link layer is sending out connection parameter changing requests, this can be sent from either of the devices.

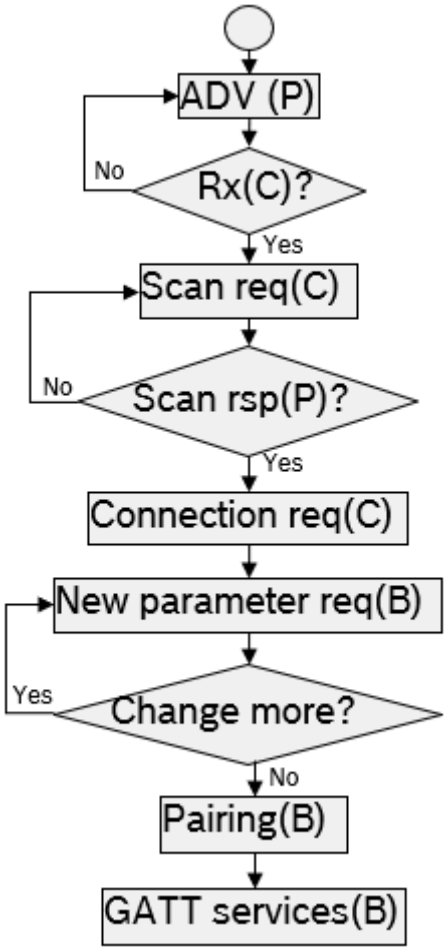

Figure 4.3: Link Layer message exchange diagram

Uncoded packet format

Uncoded packets don't contain any encryption. Most of the time IoT devices don't have encryption enabled as it makes for a more difficult process in exchanging data (e.g., a key must be exchanged). 4.4. figure shows the format of an uncoded packet.

LSB MSB

Preamble	Access Address	PDU	CRC	Const Tone Extension
1 or 2 bytes	4 bytes	2-258 bytes	3 bytes	16-160 µs

Figure 4.4: Uncoded packet

The packet in Link Layer can be separated into four segments (not including padding):

- Preamble: It is used to synchronize frequency and time between the transmitter and receiver. For 1Mbps it is 1 byte, 2Mbps 2bytes.
- Access Address (AA): As mentioned in the beginning of the section BLE uses a limited number of carrier frequencies (40). Therefore, packet collision from two or more devices in the vicinity of each other could easily happen. Which is why an access address is sent out at the beginning of the packet to distinguish connections. The default value is 10001110100010011011111011010110b (0x8E89BED6), for advertisement packets usually it stays default.
- PDU: 2-258 bytes. Contains the "useful" data.
- CRC: 3 bytes. Calculated from the PDU with a polinom to validate the message.

Chapter 5

Design and implementation

5.1 Motivation and specification

To propose a solution for the project the expectations of our tool should be defined. This tool will be used commercially by Bosch for security testing of devices without knowing much about them (only MAC address is known), thus the fuzzer should be **black-box**(3.2.1).
Previously this topic was given to other engineers at the team. They could fuzz host layer messages as written in Chapter 4 these are the layers which are easily accessible by developers, but they could not send out link layer messages directly. This was not sufficient, as link layer messages don't get much attention by application developers. Section 4.3.2 disclosed the link layer message exchange during a connection build-up. Figure 5.1 shows that these messages can't be secured because the pairing process starts after setting up the connection parameters, that is the reason why most DoS (1.2) vulnerabilities for BLE use link layer messages, so the tool should focus **on Link Layer messages** of course with capability to **expand it to other layers**.

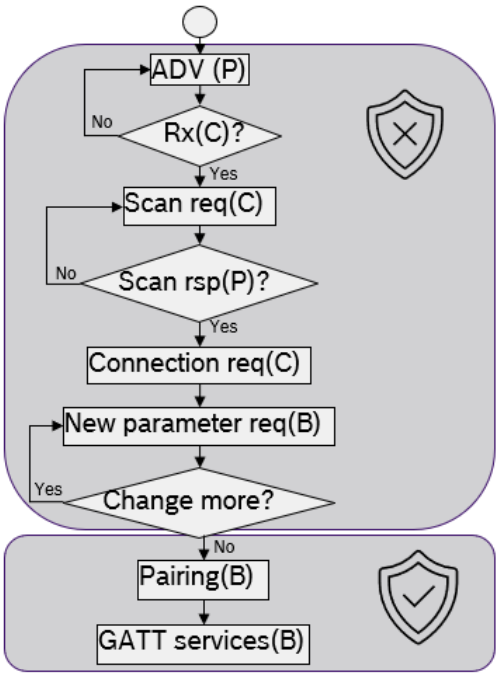

Figure 5.1: Link layer security

The length of BLE specification should already suggest the message generation approach. BLE has a finite set of message types in each layer. Other than GATT messages, the use of these messages don't differ from device to device that much. Therefore the most appropriate choice for message generation is the **generation-based** (3.2.2) approach.
Finally, the easiest choice to make is the intelligence of our fuzzer as described in Section 3.2.3 **smart** fuzzers are more efficient than their counterparts.

5.2 State of the art

During my research I encountered three major state of the art solutions for BLE fuzzing: Defensics by Synopsys[24], Sweyntooth by Matheus Garbelini[25] and Andrea Pferscher's tool[27].

Defensics is a comprehensive, versatile, automated black-box fuzzer that enables organisations to efficiently and effectively discover and remediate security weaknesses in software.[24]
Sweyntooth framework runs in a central device and tests a BLE device when the latter gets connected to the central as a peripheral. it incorporates a state machine model of the suite of BLE protocols and monitors the peripheral's state through its responses. With the state machine and current state of the central, the framework either sends malformed packets or normal packets at a wrong time, or both, to the peripheral and awaits an expected response.[25] The framework is not open-source, but proof-of-concept scripts[26] are available for the discovered vulnerabilities.
Andrea's tool provides a learning-based fuzzing framework for Bluetooth Low Energy (BLE) devices. The framework consists of two components. The first component is the learning component which learns the behavioural models of BLE devices. The second component is the stateful fuzzer which fuzz tests the BLE devices based on the previously learned model.[27] The framework is open-source but can't be used commercially.

For the project to make sense I must ask the question: why do I need to implement BLE fuzzing myself if there are working open-source examples?

- As stated, the only commercial tool out of the three is Defensics. As this tool would be used for security testing in the real world the license to these tools should be acquired which would be costly in our case.
- Integrating an already existing complex software to the test framework of the laboratory is not possible, or only with too much effort which is not acceptable due to financial reasons.
- Last but definitely not least, these problems already occurred as my tool is developed on top of an already working framework, which is a black-box, smart, mutation-based fuzzer for automotive networks by Dániel Lakatos[28]. His solution was created with modularity in mind so that the individual components of the software can be changed anytime. As the fuzzing engine is a separate component as well, it will be always possible to use an existing solution for data manipulation.

5.3 Implementation

Figure 5.2: BLE fuzzing structure

A fuzzer has three main stages (3.1): message generator,- publisher and target monitor.

5.3.1 Message generator

As the working framework [28] was developed with modularity in mind, it was relatively easy to expand it, however [28] was only used for automotive wired protocols which have a much simpler approach to data exchange (no handshake, states), thus mutation based message generation was used by it, so one big part of the development was implementing a generation-based message generation approach which was aware of the BLE states.
To be able to achieve this the valid format of each message must be precisely defined in the software. This is done by an abstract message structure, where a message is built from multiple individually accessible fields. With this design, the fuzzing engine can abstract the messages to byte strings and do its job independently from the protocol format. a field has various parameters which affect the fuzzing logic and hereby make sure that the malformed messages look as valid as possible.
The most important parameters are “fuzzable” and “variable length”, which indicate whether a field should be fuzzed at all, and if yes, is the engine allowed to modify its length or not. The type of these fields is not binary, but a float value which ranges from 0.0 to 1.0 and represents a weight for probability. [28]

The message domain I set was of the link layer messages represented in the Figure 5.3. As you can see the message domain has already been extended to L2CAP and ATT layer messages as they are also connection parameter changing messages.

The message types have their own unique message fields, so every message structure had to be implemented by hand. An example can be seen in the following code snippet:

Listing 5.1: Code snippet of a message type definition

```
class BTLE_LL_CONNECTION_UPDATE_IND(Protocol):


    def __init__(self,
        name = "BTLE_LL_CONNECTION_UPDATE_IND",
        header = BLE_LLHeader(),
        #BTLE_DATA
        RFU = BitField(value = "000", fuzzable = 0.0, variable_length = 0.0, name = "RFU"),
        MD = BitField(value = "0",fuzzable = 0.0,variable_length = 0.0,name = "MD"),
        SN = BitField(value = "0",fuzzable = 0.0,variable_length = 0.0,name = "SN"),
        NESN = BitField(value = "0011",fuzzable = 0.0,variable_length = 0.0,name = "NESN"),
        LLID = BitField(value = "0011",fuzzable = 0.0,variable_length = 0.0,name = "LLID"),
        Length = BitField(value = "00100001",fuzzable = 0.0,variable_length = 0.0,name = "Length"),
        #BTLE_CTRL
        opcode = BitField(value = "0",fuzzable = 0.0,variable_length = 0.0,name = "opcode"),
        #CHANNEL_MAP_IND
        win_size = BitField(value = bin(0)[2:], fuzzable = 0.0, variable_length = 0.0, name = "win_size"),
        win_offset = BitField(value = bin(0)[2:], fuzzable = 0.0, variable_length = 0.0, name = "win_offset"),
        interval = BitField(value = bin(6)[2:], fuzzable = 0.0, variable_length = 0.0, name = "interval"),
        latency = BitField(value = bin(0)[2:], fuzzable = 0.0, variable_length = 0.0, name = "latency"),
        timeout = BitField(value = bin(50)[2:], fuzzable = 0.0, variable_length = 0.0, name = "timeout"),
        instant = BitField(value = bin(6)[2:], fuzzable = 0.0, variable_length = 0.0, name = "instant"),
    ):
        super(BTLE_LL_CONNECTION_UPDATE_IND, self).__init__(header, name)
```

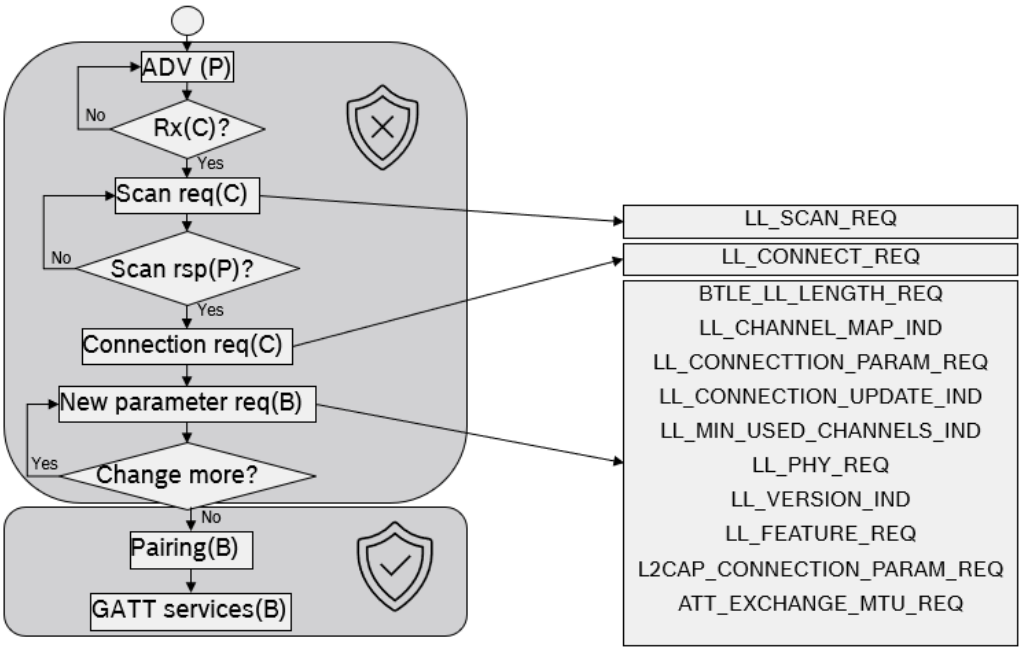

Figure 5.3: Message domain

The fields have default parameters assigned to them and 5.4. figure explains one message mutation in an illustrative manner. This will alter one field at a time.

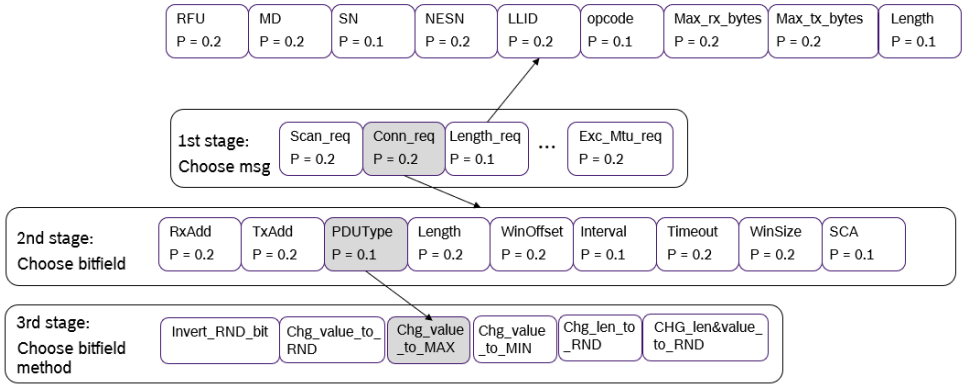

Figure 5.4: Example of message generation

As for now there are only stand-alone fields of messages, thus they have to be assembled and wrapped into an uncoded link layer message format(4.3.2). Messages are assembled and sent using Scapy[29], which is a packet manipulation toolkit written in Python. The malformed data is wrapped into a BLE format created by the Scapy library.

5.3.2 Message publisher

The malformed message is created and waiting to be sent out to the SUT by the message publisher(3.1.2). Finding an appropriate tool was hard to find for this job as the fuzzing tool was developed in Python and in most cases BLE development kits can't send out link layer messages directly. My first approach was to develop the controller layers from the ground up with a Software Defined Radio[30]. However, after a while it became apparent that this was a big undertaking as BLE uses frequency hopping which needed to be implemented on the SDR in order to achieve the connection, however the device had limitations in its speed of changing frequencies and synchronizing to it quickly enough.

Figure 5.5: PlutoSDR[31]

The next approach involved searching for similar BLE fuzzing platforms and their solutions to this problem. The Sweyntooth and Andrea's project used the same nRF5240 Dongle with custom firmware open-sourced by the Sweyntooth project, so I decided to use this as a platform for my message publisher.

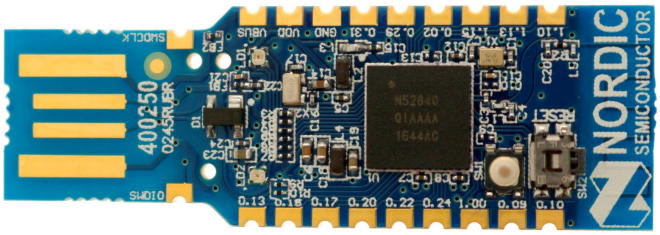

Figure 5.6: nRF5240 Dongle [32]

5.3.3 Target monitor

Finally, the last phase target monitor (3.1.3) needs to be added. As already mentioned in Section 3.3. embedded systems lack feedback, the situation evolves even worse for wireless systems. Based on the target there are two type of test setups:

- Powered by a coin-cell battery without the ability to prop open the casing of the device (e.g. smartlock with steel casing)

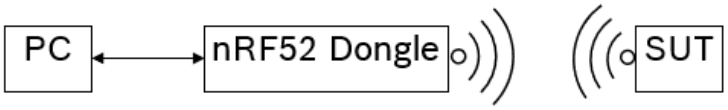

Figure 5.7: 1st type of setup

  - Health check: To decide whether the system under test has been affected by the provided input or not.
    - Cyclic message exchange check: During a connection the Slave and the Master device must exchange DATA type messages with empty PDU periodically. It's function is to listen for these type of messages. The period of the method's frequency can be modified.
  - Keep Alive: Loss of the connection to a connection timeout should be prevented with some methods.
    - SCAN REQ, CONNECTION REQ periodically: Sends out a sequence of not-fuzzed SCAN REQ and CONNECT IND messages. The period of the method's frequency can be modified.

- Access to the PCB so the device can be powered through an external power supply (e.g. iTAG).

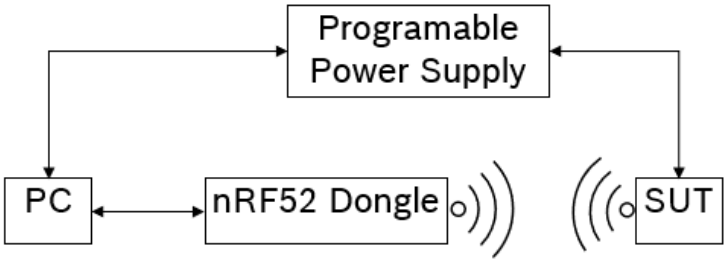

Figure 5.8: 2nd type of setup

  - Health check: To decide whether the system under test has been affected by the provided input or not.
    - Cyclic message exchange check
    - Power consumption monitor: Monitor current consumption, through which its able to detect SUT reboot.

- Keep Alive: Loss of the connection to a connection timeout should be prevented with some methods.
  * SCAN REQ, CONNECTION REQ periodically
  * Hard reset control: DoS can only be fixed though a hard reset, so for fault reproduction purposes it is vital to connect our device to a programmable power supply.

Chapter 6

Results of the execution

This section of the paper describes the effectiveness of the fuzzing tool on three IoT devices using BLE for communicational purposes. (As these are not Bosch specific products the name of these devices can be stated: OKLOK Smart Lock, iTAG, Smart Lightbulb.)

Figure 6.1: Systems under test: OKLOK Smart Lock, Philips Hue, iTAG

For the smart lock and the Lightbulb I used the first type of test setup (5.7), this setup has the advantage that the SUT doesn't need any preparation, but fault reproduction and automation is hardly possible as the tester manually has to take the battery out to cause a hard reset. The iTAG used the second (5.8), thus the iTAG PCB needed to have wires attached to the VCC and the GND so it can be powered through a programmable power supply. This was a way more powerful setup as the fuzzing process could run on its own without any interference by the tester. However, as the PCB is exposed man maid fault can occur.

In order to be able to fuzz the given BLE address of the SUT need to be discovered with a BLE sniffer (6.2). Alternative method would be discovering it with a Bluetooth capable phone, however I reviewed the fuzzed connection with the BLE sniffer during the connection.
During fuzzing, the received and sent out messages by the message publisher get logged in txt format as shown in Figure 6.3.

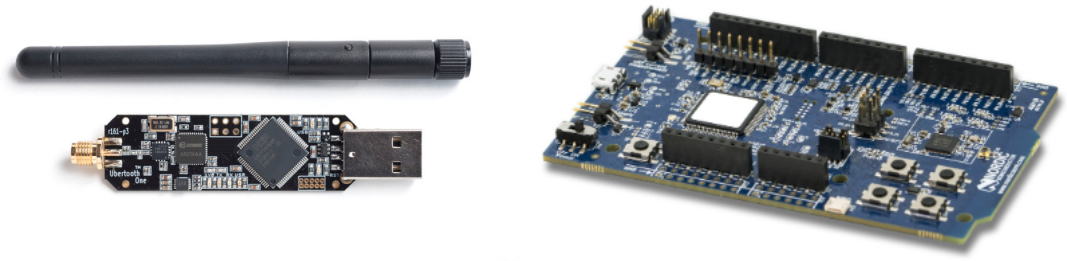

Figure 6.2: BLE sniffers: Ubertooth Project[33], nRF52 DK[34]

```
[Reset] Resetting DUT.
[Init] Executing initializer function.
[Init] The DUT is initialized.
[Test Step] Setting initial seed to 443675147442471257173941809910735000897160.
[Info] Block started.
[Test Step] Resetting seed to 739941704814005688591946486 77.
[Test Step] Chosen message: BTLE_LL_LENGTH_REQ.
[Test Step] Running iteration no. 1.
[Fuzzing] Using: bitfield_change_value_and_length_to_random.
[Fuzzing] Modifying MD value
[Fuzzing] from  0
[Fuzzing] to    1.
[Test Step] Running iteration no. 2.
[Test Step] Modifying previous message.
[Fuzzing] Using: bitfield_invert_random_bit.
[Fuzzing] Modifying SN value
[Fuzzing] from  0
[Fuzzing] to    1.
[Test Step] Running iteration no. 3.
[Test Step] Modifying previous message.
[Fuzzing] Using: bitfield_invert_random_bit.
[Fuzzing] Modifying MD value
[Fuzzing] from  1
[Fuzzing] to    0.
[Test Step] Running iteration no. 4.
[Test Step] Modifying previous message.
[Fuzzing] Using: bitfield_invert_random_bit.
[Fuzzing] Modifying max_tx_bytes value
[Fuzzing] from  00000000
[Fuzzing] to    01000000.
```

Figure 6.3: File logger

As seen in Figure 6.3 the fuzzer first resets the DUT if is in the 2nd type of setup and executes an initializer function which can be seen in the 6.1. code snippet. This function sets up a connection with the SUT.

Listing 6.1: Initiazing function

```
def func():
    # scan for device
    TE.SendBLE(BTLE() / BTLE_ADV() / BTLE_SCAN_REQ(
        ScanA=DUTParams.Scan_addr,
        AdvA=DUTParams.Adv_addr))
    scan_time_start = time.time()
    #rx data
    Connection = False
    while Connection is False:
        data = TE.RxBLE()
        pkt = BTLE(data)

        if data and (BTLE_SCAN_RSP in pkt or BTLE_ADV in pkt) and pkt.AdvA == DUTParams.Adv_addr.lower():
                DUTParams.Scan_time_delta = time.time() - scan_time_start + 1
                DUTParams.RxAdd = pkt.TxAdd

                print(DUTParams.Adv_addr.upper() + ': ' + pkt.summary()[7:] + ' Detected')
                # Send connection request to advertiser
                conn_request = BTLE() / BTLE_ADV(RxAdd=DUTParams.RxAdd, TxAdd=0) / BTLE_CONNECT_REQ(
                    InitA=DUTParams.Scan_addr,
                    AdvA=DUTParams.Adv_addr,
                    AA=DUTParams.access_addr,  # Access address (any)
                    crc_init=0x179a9c,  # CRC init (any)
                    win_size=2,  # 2.5 of windows size (anchor connection window size)
                    win_offset=1,  # 1.25ms windows offset (anchor connection point)
                    interval=16,  # 20ms connection interval
                    latency=0,  # Slave latency (any)
                    timeout=50,  # Supervision timeout, 500ms (any)
                    chM=0x1FFFFFFFFF,  # Any
                    hop=5,  # Hop increment (any)
                    SCA=0,  # Clock tolerance
                )
                TE.SendBLE(conn_request)
                Connection = True
                print(DUTParams.Adv_addr.upper() + ': ' + pkt.summary()[7:] + ' Connected')
```

Out of the three SUT two of them crashed and one deadlocked.

- Crash: Vulnerabilities in this category can remotely crash a device by triggering hard faults. This happens due to some incorrect code behaviour or memory corruption, e.g., when a buffer overflow on BLE reception buffer occurs. When a device crash occurs, they usually restart. However, such a restart capability depends on whether a correct hard fault handling mechanism was implemented in the product that uses the vulnerable BLE SoC. [25]

- Deadlock: Deadlocks are vulnerabilities that affect the availability of the BLE connection without necessarily causing a hard fault or memory corruption. Usually they occur due to some improper synchronisation between user code and the SDK firmware distributed by the SoC vendor, leaving the user code being stuck at some point. Crashes originating from hard faults, if not properly handled, can become a deadlock if the device is not automatically restarted. In most cases, when a deadlock occurs, the user is required to manually power off and power on the device to re-establish proper BLE communication.[25]

As the internal behavior of the system was completely hidden from us, I could not determine the root cause of the failure.

Chapter 7

Conclusion and further development

7.1 Conclusion

Chapter 1 shows that wireless devices can be hacked and it is important to find and fix as many vulnerabilities as possible before releasing the product. This project shows that fuzzing is a very effective security testing method and that it is easy to corrupt a device with only slight alterations from a proper message format which should be considered as a success. Some case studies are presented in Chapter 6, which were susceptible to DoS type attacks.

7.2 Future development

7.2.1 Side-channel analysis during fuzzing

3.3., 3.2.3. sections propose an intriguing behaviour: embedded device fuzzing lacks feedback, however providing feedback to fuzzed data generation makes the fuzzing tool more effective.
For now, the approach that caught my eyes would involve inspecting power consumption. Power side-channel analysis, inspects how much power certain operations consume on certain inputs. Different operations require different amount of energy so measuring the power consumption can reveal the internal operations, which lead to the theory that checking power consumption in each fuzzing cycle would also provide unique and not just general (i.e.: power up/-down) information.
Also considering that the communication uses radio frequency signals, electromagnetic side-channel analysis (examining not the power consumption, but the electromagnetic radiation) could also prove useful.

7.2.2 Handle statefulness

A great improvement of the software would be the ability to handle the behaviour of stateful protocols. This is called stateful fuzzing. With this technique, the fuzzing may reach areas of code only executed at a later stage in a protocol run, therefore increasing the probability of failures.

For instance, in the context of BLE, trivially the pairing process starts after a connection is established, so the fuzzer should be awear of this. I already considered the statefulness of BLE in section 4.3.2, for now I only implemented an initialising function (6.1) as link layer fuzzing was the main objective.

Acknowledgement

I would like to express my gratitude to my two advisors: Dr. Csilling Ákos, who guided me throughout this project, Dr. Horváth Bálint, who helped internally at the university. I would also like to thank Saulaiman Mera and Lakatos Dániel both of whom supported me and offered deep insight.

Bibliography

[1] Grant Hernandez, Orlando Arias, Daniel Buentello, Yier Jin: Smart Nest Thermostat: A Smart Spy in Your Home

[2] Estes, A.C. This Nest Security Flaw is Remarkably Dumb. Available online: https://gizmodo.com/this-nest-security-flaw-is-remarkably-dumb-1793524264 (accessed on 21.10.2022.).

[3] Nitesh Dhanjani: Hacking Lightbulbs: Suecurity Evaluation of the Philips Hue Personal Wireless Lighting System

[4] Aditya Gupta: The IoT Hacker's Handbook A Practical Guide to Hacking the Internet of Things

[5] Jay Radcliffe: Medical Devices for Fun and Insulin: Breaking the Human SCADA system available online https://paper.bobylive.com/Meeting_Papers/BlackHat/USA-2011/BH_US_11_Radcliffe_Hacking_Medical_Devices_Slides.pdf (accessed on 30.10.2022.)

[6] Anthony Rose, Ben Ramsey: Picking Bluetooth Low Energy Locks from a Quarter Mile Away. Available online: https://av.tib.eu/media/36217 (accessed on 24.10.2022.)

[7] https://uk.pcmag.com/smart-thermostats/129928/nest-thermostat (accessed on 30.10.2022.)

[8] https://www.lampenwelt.de/philips-hue-white-color-ambiance\-6-5w-e27-2er-set.html (accessed on 30.10.2022.)

[9] https://www.diabetesnet.com/diabetes-technology/insulin-pumps/older-pumps/onetouch-ping/ (accessed on 30.10.2022.)

[10] https://de.manuals.plus/oklok/fb50-bluetooth-and-fingerprint-lock-manual (accessed on 30.10.2022.)

[11] Sultan Khan: Technical Advisory – Tesla BLE Phone-as-a-Key Passive Entry Vulnerable to Relay Attacks. Available online: https://research.nccgroup.com/2022/05/15/technical-advisory-tesla-ble\-phone-as-a-key-passive-entry-vulnerable-to-relay-attacks/ (accessed on 24.10.2022.)

[12] Dr. Charlie Miller,Chris Valasek: Remote Exploitation of an Unaltered Passenger Vehicle

[13] Dr. Charlie Miller,Chris Valasek: Remote Exploitation of an Unaltered Passenger Vehicle

[14] Vaibhav Bedi: The Practical Guide to Hacking Bluetooth Low Energy. Available online: https://blog.attify.com/the-practical-guide-to-hacking-bluetooth-low-energy/ (accessed on 24.10.2022.)

[15] Stephanie Bayer, Alexander Ptok: Don't Fuss about Fuzzing: Fuzzing Controllers in Vehicular Networks

[16] Xiaotao Feng, Ruoxi Sun, Xiaogang Zhu: Snipuzz: Black-box Fuzzing of IoT Firmware via Message Snippet Inference

[17] Maialen Eceiza , Jose Luis Flores , and Mikel Iturbe: Fuzzing the Internet of Things: A Review on the Techniques and Challenges for Efficient Vulnerability Discovery in Embedded Systems

[18] V. J. M. Manes et al., "Fuzzing: Art, science, and engineering," Nov. 2018.

[19] B. Zhao, J. Li, and C. Zhang, "Fuzzing: A survey," Cybersecurity, vol. 1, p. 6, Jun. 2018.

[20] G. J. Saavedra, K. N. Rodhouse, D. M. Dunlavy, and P. W. Kegelmeyer, "A review of machine learning applications in fuzzing," Jul. 2019.

[21] B. Liu, L. Shi, Z. Cai, and M. Li, "Software vulnerability discovery techniques: A survey," in Proc. 4th Int. Conf. Multimedia Security, Nov. 2012, pp. 152–156.

[22] Bluetooth Core Specification v5.1 Vol. 6

[23] https://www.rfwireless-world.com/Terminology/BLE-Protocol-Stack-Architecture.html (accessed on 30.10.2022.)

[24] https://www.synopsys.com/software-integrity/security-testing/fuzz-testing.html (accessed on 30.10.2022.)

[25] Matheus E. Garbelini : SweynTooth: Unleashing Mayhem over Bluetooth Low Energy

[26] https://github.com/Matheus-Garbelini/sweyntooth_bluetooth_low_energy_attacks (accessed on 30.10.2022.)

[27] https://git.ist.tugraz.at/apferscher/ble-fuzzing

[28] Lakatos Dániel: Security Testing of Vehicle Controllers. Available online: https://diplomaterv.vik.bme.hu/hu/Theses/Jarmuipari-vezerlok-biztonsagi-tesztelese (accessed on 30.10.2022.)

[29] https://scapy.readthedocs.io/en/latest/ (accessed on 30.10.2022.)

[30] Kepics János: A Bluetooth Low Energy (BLE) protokoll sérülékenységének vizsgálata a fizikai rétegben (önálló laboratórium 1)

[31] https://www.analog.com/en/design-center/evaluation-hardware-and-software/evaluation-boards-kits/adalm-pluto.html (accessed on 30.10.2022.)

[32] https://www.nordicsemi.com/Products/Development-hardware/nrf52840-dongle (accessed on 30.10.2022.)

[33] https://greatscottgadgets.com/ubertoothone/ (accessed on 30.10.2022.)

[34] nRF52 DKDevelopment kit for Bluetooth LE, Bluetooth mesh, ANT and 2.4 GHz applications datasheet reachable from https://www.nordicsemi.com/Products/Development-hardware/nrf52-dk (accessed on 30.10.2022.)

List of Figures

1.1 From left to right: Nest Thermostat[7], Philis Hue[8], OneTouch Ping[9], OKLOK Smart Lock[10] . . . 7

2.1 Overview of embedded security evaluation . . . 9

3.1 Structure of fuzz testing . . . 12
3.2 Taxonomy of fuzzing . . . 13

4.1 BLE stack . . . 18
4.2 Link layer state graph . . . 20
4.3 Link Layer message exchange diagram . . . 21
4.4 Uncoded packet . . . 21

5.1 Link layer security . . . 23
5.2 BLE fuzzing structure . . . 25
5.3 Message domain . . . 26
5.4 Example of message generation . . . 26
5.5 PlutoSDR[31] . . . 27
5.6 nRF5240 Dongle [32] . . . 27
5.7 1st type of setup . . . 28
5.8 2nd type of setup . . . 28

6.1 Systems under test: OKLOK Smart Lock, Philips Hue, iTAG . . . 30
6.2 BLE sniffers: Ubertooth Project[33], nRF52 DK[34] . . . 31
6.3 File logger . . . 31

List of Tables

4.1 Bluetooth Classic and BLE 17