

# INTERNET OF THINGS

REPORT BY CHIARA SIMONI AND  
GIADA MARTINA STIVALA

## Some project on debugging Bluetooth Low Energy devices

A.A. 2017/2018  
Prof. Luciano BONONI, Marco DI FELICE

July 22, 2018



# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>Bluetooth Low Energy Specification</b>	<b>3</b>
2.1	Network Topology and Devices . . . . .	3
2.1.1	Decypher Advertising Packets . . . . .	4
	<b>Bibliography</b>	<b>7</b>
<b>A</b>	<b>GATT Assigned Numbers</b>	<b>9</b>



# Chapter 1

## Introduction

Bluetooth Low Energy devices are flooding the market of cheap, wearable, health, home-appliance electronics. Some examples include monitoring devices for sport, health and outdoor activities in general; they also connect many household appliances as well as security cameras and locks. Other interesting use-cases include wristbands used in theme parks or live music concerts (although the technology may differ).

Intrinsically, these devices perform very simple tasks, mostly in the field of data gathering. It seems totally harmless to have a smart watch tracking your position at any time of day, but what if it were possible for anyone to query it and extract its data? Their minimalism is what makes them so appealing to the general public, but it is sometimes their intrinsic weakness.

Moving a bit further from "*hackers*" and information security, data gathering is a fundamental step for data analysis in a Big Data context. Privacy concerns should be taken into consideration also in terms of data anonymization.

In this project report, we will start by describing general device characteristics and Bluetooth specifications. We will then move to their concrete observation with hands-on analyses, taking into consideration different developing tools and devices. In particular, we will focus our attention on security risks and concrete flaws, highlighting security exploits when possible. Finally, we will summarize our findings, including a few ideas for future directions and developments.

For our tests, we used a Mi Band 2 smart band, a Magic Blue smart light-bulb and a ST Microelectronics IoT node (model `STM32L475 MCU`). The code was run both on Ubuntu 16.04 and Kali Linux 4.13. It is worth mentioning that we also tried to use a Ubuntu GNOME OS, virtualized via VMWare on Windows 10, but this option was soon discarded as VMWare does not provide the drivers for Bluetooth Low Energy.

The software at our disposal included the builtin Linux commands, as well as Wireshark and Android apps (namely BLEScanner and nRFCon-

## 1. Introduction

---

nect). As we will see, we also tested some open-source tools available from Github: bleah, btlejuice and gattacker.

## Chapter 2

# Bluetooth Low Energy Specification

In this chapter we briefly report the main technical characteristics of the Bluetooth Low Energy Personal-Area-Network technology. Our main reference is the Core Specification 5.0 available on the official website. When possible, we couple the theoretical concepts with some examples from our own experience.

### 2.1 Network Topology and Devices

Devices can be divided into three categories: Advertisers, Scanners and Initiators. Communication has to be started by an initiator device following an *advertisement connectable packet* (ADV\_IND). Advertisement is performed on the primary channels, while bidirectional communication happens on one of the 37 secondary channels, decided during the connection procedure. When a connection is established, the initiator and the advertiser respectively become the master and slave devices; while the former can be involved in more than one communication, the slave device can belong to a single *piconet* at a time. In fact, it is not possible to have a BLE device paired with more than one "master" device at the same time.

As we had the possibility to concretely program a BLE device, we show in Listing 2.1 an extract of code in which the device is advertising *connectable* packets.

Listing 2.1: Set up flags for device mode of operation: connectable

```
1 ble_gap().accumulateAdvertisingPayload(GapAdvertisingData::  
    BREDR_NOT_SUPPORTED | GapAdvertisingData::  
    LE_GENERAL_DISCOVERABLE);  
2 ble_gap().accumulateAdvertisingPayload(GapAdvertisingData::  
    COMPLETE_LIST_16BIT_SERVICE_IDS, (uint8_t *)uuid16_list,  
    sizeof(uuid16_list));
```

## 2.1 Network Topology and Devices 2. Bluetooth Low Energy Specification

```
3 ble.gap().accumulateAdvertisingPayload(GapAdvertisingData::  
    THERMOMETER_EAR);  
4 ble.gap().accumulateAdvertisingPayload(GapAdvertisingData::  
    COMPLETE_LOCAL_NAME, (uint8_t *)DEVICE_NAME, sizeof(  
    DEVICE_NAME));  
5 ble.gap().setAdvertisingType(GapAdvertisingParams::  
    ADV_CONNECTABLE_UNDIRECTED);  
6 ble.gap().setAdvertisingInterval(1000); /* 1000ms. */  
7 ble.gap().startAdvertising();
```

Advertising may also be *non-connectable* (ADV\_NONCONN\_IND): the LE device periodically sends its data on the main channel for every scanner to read. This is really convenient in terms of implementation, but it also represent the least secure solution, as data is sent in clear text. In Listing 2.2 we highlight the different parameter used to set *non-connectable* advertising.

Listing 2.2: Set up flags for device mode of operation: non connectable

```
1 ble.gap().setAdvertisingType(GapAdvertisingParams::  
    ADV_NON_CONNECTABLE_UNDIRECTED);
```

On the whole, the Specification defines six different advertising packets, among which we also report the request for additional information from the advertisement (SCAN\_REQ), it having a direct corresponding bash command.

### 2.1.1 Decypher Advertising Packets

Before considering Man-In-The-Middle techniques, we show how it is possible to intercept data from advertising devices, mentioned in the previous section. We have to consider that devices advertise at random intervals to avoid collisions, thus it may happen that the target device is not immediately discovered by the scanner.

For this experiment, we used the STM IoT node and tested it on both our Linux distributions. The board retrieves the temperature and humidity levels in the environment and broadcasts them as a non-connectable advertising packet, as we can see in Listing 2.3.

Listing 2.3: Retrieve sensor values and update packet

```
1 uint8_t temp = static_cast<unsigned int>(sensor_value);  
2 uint8_t hum = static_cast<unsigned int>(humidity_value);  
3  
4 uint8_t service_data[4];  
5 service_data[0] = GAPButtonUUID & 0xff;  
6 service_data[1] = GAPButtonUUID >> 8;  
7 service_data[2] = temp;  
8 service_data[3] = hum;  
9  
10 BLE::Instance().gap().updateAdvertisingPayload(  
    GapAdvertisingData::SERVICE_DATA, (uint8_t *)service_data,  
    sizeof(service_data));
```



Packets broadcasted by the board can be retrieved with any of the tools described at the end of the previous chapter. We analysed the payload and found that it follows this structure:

*length - meaning - content.*

The length is expressed in bytes and it refers to the overall size of *meaning* and *content*. The former two is a label which has to be matched with the corresponding Bluetooth GAP Assigned Number (specification available on the website, we report the table in Appendix A) to understand its meaning. Common examples are 0x09 for "Complete Local Name" or 0x0a for "Power Level". In our case, the label corresponds to "Service data 16-bit UUID", which means that we should look at the next two bytes to have a more precise indication of the received data. Vendors usually publish a web page in which they list all their codes with the respective meaning, while in this example we can see that its value is added to the payload in `service_data[0]`, and corresponds to 00 aa.

Finally, the next two bytes encode the temperature and humidity in the room.

## 2.2 Connection

When the Scanner device answers to an advertisement message, a secondary channel is established for bidirectional communication. This channel is randomly chosen and communicated to the advertiser in the connection request packet. As previously mentioned, the slave device can be paired with a single master at a time, consequently all subsequent communications will happen in the established secondary channel, also granting additional connection speed.

Pairing procedures vary depending on the Bluetooth Low Energy device. They are mentioned in the Specification as *Association modes* and they are the following:

- Numeric Comparison: used when the devices can show numbers (at least 6 digits) and can receive user input (e.g. yes or no), like a phone or a laptop. Devices wishing to pair should show the same number, and the owners have to confirm the match.
  - This usually happen when pairing two mobile phones or a mobile phone to a laptop. In case the BLE device has a display, this association mode is attempted.
- Just Works: used when 1+ devices doesn't have a keyboard for user IO and / or cannot display 6 digits. Consequently, no PIN is shown and the user just has to accept the connection.

- This is the case of the STM IoT Node. When attempting to connect from any Scanner device, the connection is established without authentication.
- It is also the case of the Magic Blue smart bulb: the device pairs with any scanner.
- In both situations, to unpair from the device it is necessary to unplug from the power source.
- Out Of Band: used when there is a (secure) OOB channel for pairing and security keys exchange. Of course, if such channel is not secure the whole process may be compromised.
- Passkey Entry: one device has input capabilities, but can't display 6 digits; the other device has output capabilities: the PIN is showed on the second device, and the connection is "confirmed" by entering the same PIN on the first device. Note that in this case the PIN is created by a specific security algorithm, while in legacy versions was an input from the user.
  - This is the case of the Mi Band 2 smart band, in which it is not possible to display PIN numbers, but there is an input device: when attempting to pair, the band vibrates and asks for confirmation by tapping its button.

# Bibliography



## Appendix A

### GATT Assigned Numbers

Data Type	Data Type Name	See advlib section
0x01	Flags	Flags
0x02	Incomplete List of 16-bit UUIDs	UUID
0x03	Complete List of 16-bit UUIDs	UUID
0x04	Incomplete List of 32-bit UUIDs	UUID
0x05	Complete List of 32-bit UUIDs	UUID
0x06	Incomplete List of 128-bit UUIDs	UUID
0x07	Complete List of 128-bit UUIDs	UUID
0x08	Shortened Local Name	Local Name
0x09	Complete Local Name	Local Name
0x0a	Tx Power Level	Tx Power
0x0d	Class of Device	Generic Data
0x0e	Simple Pairing Hash C-192	Generic Data
0x0f	Simple Pairing Randomizer R-192	Generic Data
0x10	Security Manager TK Value	Generic Data
0x11	Security Manager OOB Flags	Generic Data
0x12	Slave Connection Interval Range	SCIR
0x14	16-bit Solicitation UUIDs	Solicitation
0x15	128-bit Solicitation UUIDs	Solicitation
0x16	Service Data 16-bit UUID	Service Data
0x17	Public Target Address	Generic Data
0x18	Random Target Address	Generic Data
0x19	Public Target Address	Generic Data
0x1a	Advertising Interval	Generic Data
0x1b	LE Bluetooth Device Address	Generic Data
0x1c	LE Bluetooth Role	Generic Data
0x1d	Simple Pairing Hash C-256	Generic Data
0x1e	Simple Pairing Hash Randomizer C-256	Generic Data
0x1f	32-bit Solicitation UUIDs	Solicitation
0x20	Service Data 32-bit UUID	Service Data
0x21	Service Data 128-bit UUID	Service Data
0x22	LE Secure Con. Confirmation Value	Generic Data
0x23	LE Secure Connections Random Value	Generic Data
0x24	URI	Generic Data
0x25	Indoor Positioning	Generic Data
0x26	Transport Discovery Data	Generic Data
0x27	LE Supported Features	Generic Data
0x28	Channel Map Update Indication	Generic Data
0x29	PB-ADV	Generic Data
0x2a	Mesh Message	Generic Data
0x2b	Mesh Beacon	Generic Data
0x3d	3-D Information Data	Generic Data