Bachelor's Thesis Computing Science

Fuzzing Zigbee using Z-Stack

Tom Rust
s1040068

June 10, 2022

First supervisor/assessor:
Dr. Ir. Erik Poll

Second assessor:
Dr. Katharina Kohls

Radboud University

Abstract

Nowadays, a lot of houses contain smart devices. While they can be a blessing for their users, they can also become a weak spot in their home security. This weakness typically sits in their communication interface, common forms of communication are Bluetooth (LE), Wi-Fi, Z-Wave and Zigbee. Zigbee is based on top of the IEEE 802.15.4 standard and is most commonly used as a way of communication for smart home devices, such as smart light bulbs or thermostats. This bachelor thesis will focus on breaking the security of Zigbee using a technique called fuzzing.

Fuzzing is a way of testing the implementation of the communication interface on the target device. With fuzzing, you typically send a lot of data to the target device, a common example is setting the package length to 0 and sending a lot of data. In this case, fuzzing is applied to find a mistake in the Zigbee implementation of a Philips Hue light bulb.

The results show that using a Zigbee adapter running the Z-Stack firmware it is possible to fuzz most fields of the Application layer of the Zigbee protocol and some fields of the Network layer and the MAC layer. A second Zigbee adapter operating as a Zigbee sniffer is required to verify which fields can actually be fuzzed. It was found that fuzzing other fields is very difficult because the Z-Stack firmware running on the Zigbee adapter takes care of a large part of the Zigbee Network Protocol Stack. In the end, no bugs in the Zigbee implementation of the Philips Hue light bulb were discovered.

Contents

Chapter 1

Introduction

1.1 Motivation

Zigbee connected devices are installed in many households. For the inhabitants' sake, it is important that these devices do not become a weak spot in their security. However, Zigbee has much more possibilities: In the United States, it is also used to gather energy data on a much larger scale[1]. Furthermore, Zigbee may also be used in Wireless Body Area Networks[19]. This means that all sorts of sensors are integrated in your body, measuring motion, temperature, heart rate and thus overall health. In any case, breaking the security of Zigbee can have nasty consequences.

1.2 Research Question

The goal of this thesis is finding a method to test the implementation of the network interface of Zigbee devices. So with this in mind, the research question is:

Can we find bugs in Zigbee connected devices using fuzzing?

This comes with some sub questions:

1. Can we send Zigbee messages in any order?
2. Can we send fully customized Zigbee messages?
3. How low on the network stack can we send messages?

1.3 Content

This thesis is focused on testing the security of Zigbee connected devices. In the Chapter 2, the basics concepts of this thesis will be explained. This

[1]`https://hackaday.com/2022/01/13/remoticon-2021-hash-salehi-outsmarts-his-smart-meter/`

includes a comprehensive guide to the Zigbee protocol in section 2.1 and an explanation of the security aspect of Zigbee. Since fuzzing is an important part of this research, section 2.2 is devoted to explain the concepts and techniques of fuzzing. Section 2.3 is about the hardware related aspect of this research. Our main focus is on using the Z-Stack API to fuzz. An introduction to the Z-Stack is given in section 2.4. With all the background explained, Chapter 3 will include the experiments. These experiments will be based on the research question. Chapter 6 will review the outcome and give a conclusion.

Chapter 2

Preliminaries

This chapter is devoted to explaining the background information required to understand the experiments in chapter 3. In section 2.1 we cover the theoretical side of Zigbee. Fuzzing is covered in section 2.2 and in section 2.3 we introduce the hardware that is used during the experiments. Finally, in section 2.4 we give an introduction to the Z-Stack API.

2.1 Zigbee

In this section the theoretical side of Zigbee will be explained. We begin in section 2.1.1 with the general concept of Zigbee and then we will go into the network layers of Zigbee in section 2.1.2. In section 2.1.3 we will cover the security related part of Zigbee.

2.1.1 General concept

Zigbee devices can be divided up into 3 different types:[5]

1. The coordinator: Coordinators, also known as PAN coordinators, are the smartest nodes in the network. There can only be one coordinator and it is tasked with managing connections, keys and commands. Coordinators should never be battery powered.

2. A router is typically a Zigbee device that is connected to power and has a specific function, such as operating a light. It may also serve as a node in a path to another node, so other lights may talk to this light in order to talk to the coordinator. Hence, it should be always listening and not be battery powered.

3. An end node is similar to a router, but it only communicates with its direct parent. Furthermore, the information stream is from the end node to its parent, this means the end node does not need to be

listening the entire time and can be battery powered. Typical usages of battery powered end nodes are sensors or simple buttons.

Personal Area Network

PAN or Personal Area Network is the Zigbee equivalent of a Local Area Network (LAN) for the Internet. However, Zigbee devices are typically not required to connect to something outside their PAN. However, there are so-called PAN bridges to connect to other Personal Area Networks[10]. When two Personal Area Networks are physically in close proximity. They should be on another channel to avoid interference. Hence, one of the PAN coordinators will then decide to switch.

2.1.2 Network Layers

As stated before, the Zigbee protocol is based on the IEEE 802.15.4 protocol. While the IEEE 802.15.4 takes care of the MAC layer and physical layer, Zigbee takes care of the higher layers and the security (see figure 2.1).

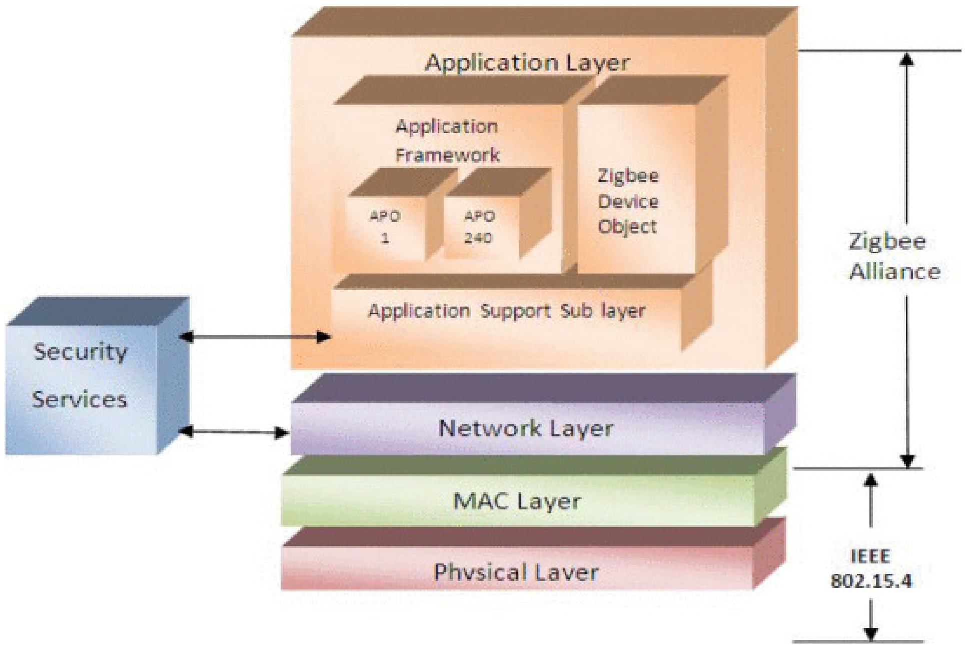


Figure 2.1: Zigbee Network Protocol Stack[17]

The MAC (Medium access control) layer is the first layer that is interesting to us. This layer is very similar to the equally named MAC layer in the OSI network model[1]. In Zigbee, there can be two different modes of transmission[26]. Beacon-enabled and non-beacon-enabled communication.

[1]https://en.wikipedia.org/wiki/Medium_access_control

With beacon-enabled communication, the coordinator broadcasts beacons, when other devices receive such a beacon, they will send their data frame back. In comparison, with non-beacon-enabled communication, the devices look to see if the channel is idle, if so, then they will transmit. Otherwise, they will wait and then look again. The MAC frames consist of a header, payload and tail. In the header the destination and source are defined, along with control information and a serial number.

Network Layer. As explained before, Zigbee devices can function as a mesh network. This is implemented in the Network Layer[24]. Services such as devices joining and leaving, addressing and neighbour and route discovery are also implemented in this layer.

The highest defined layer is the Application layer[17]. This layer itself is split up into 3 different sub-layers:

1. The application objects (APO): This layer controls the hardware and can control up to 240 application objects. An application object can for example be an LED or a sensor.

2. Zigbee device object: This part of software keeps track of the Zigbee related operations: service delivery, security and binding.

3. Application support sub layer: This layer forms an interface for the Network and Application layer. Upper application layers can use services such as: key establishment, key transport, device removal, key request, key change, entity authentication and permissions configuration table.

More detailed information about the network layers can be found in section 3.4.4 where we will inspect captured Zigbee messages.

2.1.3 Security

The security of Zigbee is based on a 128-bit AES algorithm and offers the following features:[12][16]

- Freshness: Protection against replay attacks.

- Frame integrity checking: Protection against data being modified by parties without the key.

- Entity authentication service: Allow devices to communicate with a shared key.

- Data encryption: Encrypt data by using a symmetric cipher with a key shared by 2 or more devices.

All these features are implemented in the Network layer and result in the Application Layer benefiting from these features. Zigbee relies on a Trust Center, this takes care of allowing new devices, maintaining and distributing keys and managing key configuration. Typically, this is done by the coordinator. Zigbee uses 3 different keys:

- The network key: This key is shared between all devices on the network. This can be obtained through key transport or it can be preinstalled.
- The master key: This key should be unique for each device and can be used as an initial shared secret when link keys are generated.
- Link keys: Link keys are shared between two devices, these are typically used in general practice.

2.2 Fuzzing

This section explains the concept of fuzzing, this will be done first in section 2.2.1, then we will make a distinction between black box and white box fuzzing in section 2.2.2. This is followed by some examples of fuzzing on other protocols, such as Wi-Fi in section 2.2.3. In section 2.2.4 we will look into possible issues when fuzzing Zigbee.

2.2.1 General concept

Sutton et al.[20] claims that the term fuzzing is not really existent in mainstream vocabulary in 2007. Nowadays, it is quite mainstream in security analysis but less well-known in other parts of Computing Science. Fuzzing can be seen as closely related to boundary value analysis. Which is a form of testing where we try edge cases[9], so test cases will be on the boundary of what is an allowed value or very close to it on either side. Fuzzing is defined as a method for discovering faults in software by providing unexpected input and monitoring for exceptions. This is a rather generic definition but it conveys the general idea. Key parts of the definition here are that we provide unexpected input, which is similar to boundary value analysis. This input includes input that does not adhere to the standard[13] or uses a part of the standard that is not regularly used, another example is setting the length value to 0 and still sending a lot of data. The second part is monitoring the target device or piece of software for exceptions or undefined behaviour. With software, this can often easily be done using logs.

2.2.2 Black box and white box

The concept of black box and white box is easily explained by looking at black box and white box testing. White box testing[15] is mainly done by the programmers that make the code and thus have access to it. This typically consists of unit tests[23]. In black box testing, the tester only has access to the functions the program exposes, without knowing what is exactly going on. So with a white box, the tester knows what is going on, whereas with a black box, he does not know what happens. When we apply this to fuzzing, we can define black box[25] as fuzzing a program with a black box. So we only have access to a program and its outcome. More interesting is a white box fuzzer, these are typically based on a grammar[7] and generate their data inspired by a program analysis. White box fuzzers are however much more demanding.

There also exists a combination of black box and white box fuzzing, namely grey fuzzing[2]. This is a bit in between, there is certain information available about the program, but a full analysis is not possible.

2.2.3 Examples

Fuzzing is a technique that is mainly applied to software directly, for example, in this article[6] they explain that it can be used to test websites easily for malicious data. It is also shown that this can prove very valuable for software testing.

Fuzzing software on other devices is less theoretical and more practical, it is almost always a form of black box fuzzing, because otherwise you would simply run the program on your own computer. Some researchers have successfully used a fuzzing attack on a Wi-Fi connected UAV[8]. Here they send a very large JSON(JavaScript Object Notation) object which triggers a buffer overflow. This gives an idea of the scale of possibilities that fuzzing gives.

2.2.4 Fuzzing Zigbee

Fuzzing in general is mostly done on communication implementations, since Zigbee is relatively new and runs on devices that are typically accessible from the street, it makes for a good target. However, due to the fact that Zigbee is only accessible using dedicated non-standard hardware, it may be difficult to fuzz the lower levels of the Zigbee protocol stack[18].

2.3 Zigbee Hardware

In this section the hardware related part of Zigbee will be explained. This begins with a part on Zigbee Adapter A and why it was chosen in section 2.3.1. Followed by a small part on Zigbee Adapter B in section 2.3.2. In section 2.3.3 we will look at the target device, which in our case is a Philips Hue light bulb. We will also look at the part of software that is closest to the hardware in section 2.3.4, which is Zigbee2MQTT[11]

2.3.1 Zigbee Adapter A: Sonoff Zigbee 3.0 USB

The adapter chosen for this is the *Sonoff Zigbee 3.0 USB dongle plus*, see Figure 2.2. This seems like a good choice, because it has the new CC2656P chipset[2]. This adapter is widely available throughout the world, also at redistributors in the Netherlands[3]. This adapter is able to have firmware flashed to it, which we do in section 3.1.

Figure 2.2: Sonoff Zigbee 3.0 USB dongle plus

Software

This adapter will run the Z-stack firmware from Texas Industries. This means that the adapter implements most of the Zigbee stack. We can interface this using the Z-Stack API[21]. More information on the Z-Stack API can be found in section 2.4. We will write this firmware to the adapter in section 3.1.

2.3.2 Zigbee Adapter B: CC2531

In order to gain a proper insight at Zigbee level, we will use a second adapter to sniff Zigbee traffic. We use the *Zigbee CC2531 USB-dongle* from Itead[4], see Figure 2.3. This adapter uses the CC2531 chipset from Texas Industries. This chipset is less advanced than the CC2656P from section 2.3.1, but it is

[2]`https://www.ti.com/product/CC2652P`
[3]`https://opencircuit.nl/product/sonoff-zigbee-3.0-usb-dongle-plus`
[4]`https://itead.cc/product/cc2531-usb-dongle/`

capable enough to run sniffing firmware, see section 2.3.2. This adapter is widely available throughout the world, also at redistributors in the Netherlands[5]. Technically, we could use either adapter for sniffing or fuzzing purposes. However, there are working examples of sniffing using Adapter B and it would probably not benefit from the more powerful chipset that Adapter A has.

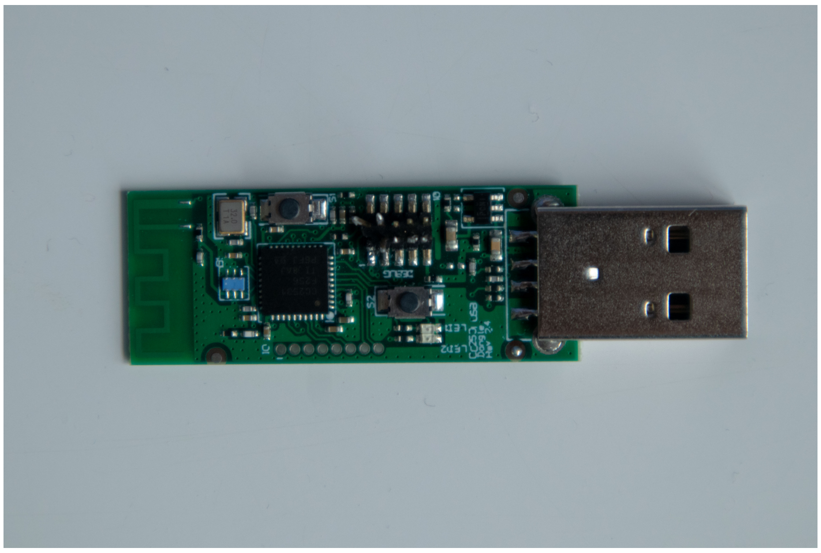


Figure 2.3: Zigbee CC2531 USB-dongle

Software

This adapter will run the PACKET–SNIFFER firmware from Texas Industries[6]. We will write this firmware to the adapter in section 3.4.

2.3.3 Philips Hue

The devices that we fuzz are Philips Hue light bulbs, see Figure 2.4. These light bulbs are very popular and installed in many households in the Netherlands. There is no particular reason that these are chosen above other Zigbee light bulbs, since the experiments could also be done with other Zigbee connected devices. However, these Philips Hue Light bulbs are available at the New Devices Lab in the Mercator.

[5] `https://opencircuit.nl/product/zigbee-cc2531-usb-dongle`
[6] `https://www.ti.com/tool/PACKET-SNIFFER`

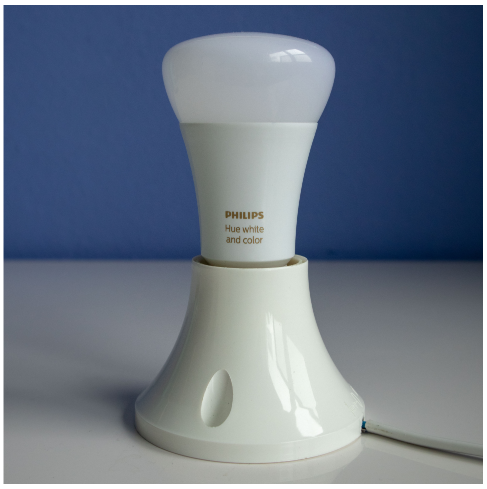


Figure 2.4: Philips Hue white and colour

2.3.4 Zigbee2MQTT

The software that provides an interface for the user to interact with Zigbee devices is Zigbee2MQTT. First the protocol MQTT will be explained and then we generally explain how Zigbee2MQTT works.

MQTT

MQTT stands for Message Queuing Telemetry Transport[7]. It is a lightweight network protocol which functions using the Publish–subscribe pattern[8]. This means that MQTT consist of a broker and multiple clients. The broker is the main node which is in the centre of all communication. These clients subscribe or publish to topics at the broker. If something changes, then clients subscribed to the topic will be notified.

Zigbee2MQTT

Zigbee2MQTT exposes the Zigbee network to MQTT, it does this internally by using the Zigbee Herdsman library[9]. There also is a useful web interface which basically acts as an MQTT client. We will use Zigbee2MQTT to learn more about communicating with Zigbee devices. Zigbee2MQTT is open source[10].

[7]https://en.wikipedia.org/wiki/MQTT
[8]https://en.wikipedia.org/wiki/Publish-subscribe_pattern
[9]https://github.com/Koenkk/zigbee-herdsman
[10]https://github.com/Koenkk/zigbee2mqtt

2.4 Z-Stack API

The Z-Stack API is the interface for the firmware that is running on Adapter A. It uses the serial port for communication. The documentation, which is both available online[11] and in a manual[21] contains all information. Some interesting aspects will be explained. In particular, the commands for the following operations:

1. Ping.
2. Transmitting data, both inside the PAN and outside the PAN.

We will use the term serial frame to discuss one transmission, which corresponds with a single command.

2.4.1 General Layout

In Figure 2.5 we can see the general layout of all serial frames. Any frame will begin with the SOF (0xfe). This is followed by the length of the bytes between Command and FCS. After this, there are two bytes containing the command, followed by the corresponding data for this command. At the end of every frame we have a checksum (FCS), which is the XOR of all bytes but the SOF. (see section 2.1.1 of the Z-Stack API)

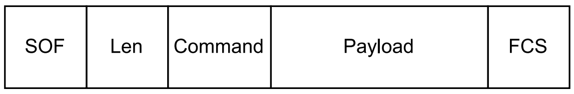


Figure 2.5: Serial frame general layout

2.4.2 SYS_PING

The SYS_PING serial frame is displayed in Figure 2.6. This can be used to confirm that the serial connection is working. This means that this serial frame will not result in a Zigbee message. To send a ping, we set the command to (0x21 0x01). The frame will then look like: 0xFE 0x00 0x21 0x01 0x20. (see section 2.2 of the Z-Stack API)

2.4.3 AF_DATA_REQUEST

The AF_DATA_REQUEST command (displayed in Figure 2.7) can be used to send messages to the application framework of another device. It is for

[11]`https://software-dl.ti.com/simplelink/esd/simplelink_cc26x2_sdk/2.30.00.34/exports/docs/zstack/html/zigbee/mt_interface.html`

SOF	Len	CMD = 0x2101	FCS

Figure 2.6: Serial frame SYS_PING

example used by Zigbee2MQTT to control the Philips Hue light and for example change colour.

This command requires the command to be set to 0x24 0x01. Furthermore, parameters need to be set for: Destination Address, Destination Endpoint, Source Endpoint, ClusterId, TransId, Options, Radius, Len and data. Most notably are the ClusterId and data. Where the ClusterId specifies what type of command it is, such as LightOnOff or ChangeBrightness. The data actually contains a payload for the Application Framework. We will be sniffing this command in section 3.3.2 and fuzzing it in 3.5.3. More information can be found in section 3.2.1.1 of the Z-Stack API.

CMD = 0x2401	DstAddr	Dst-Endpoint	Src-Endpoint	Cluster	Trans	Options	Radius	Len	Data

Figure 2.7: Serial frame AF_DATA_REQUEST

2.4.4 AF_DATA_REQUEST_EXT

According to the Z-Stack API, this command can be used to send an inter-pan message. In other words, we can use it to communicate with devices on other networks. The corresponding command is 0x24 0x02. It requires the following parameters: Destination address mode, Destination Address, Destination Endpoint, Destination Pan Id, ClusterId, TransId, Options, Radius, Message Length and Message (see Figure 2.8). Destination address mode can be set to 0x02 for a short notation or 0x03 for a long notation. We will fuzz using this command in experiment 3.5.4 and more information can be found in section 3.2.1.3 of the Z-Stack API.

CMD = 0x2402	DstAddr-Mode	DstAddr	Dst-Endpoint	PanId	Src-Endpoint	Cluster	Trans	Options	Radius	Len	Data

Figure 2.8: Serial frame AF_DATA_REQUEST_EXT

Chapter 3

Experiments

The ultimate goal of these experiments is to fuzz on a Zigbee level. To accomplish this, we can either use Zigbee Adapter A (see section 2.3.1) or Zigbee Adapter B (see section 2.3.2). We use Zigbee Adapter A for fuzzing purposes, because it is much more powerful and supports the newest Zigbee 3.0 standard. Furthermore, we can use Zigbee Adapter B for sniffing packages between Zigbee Adapter A and the SUT. Our system under test (SUT) is a Philips Hue Light (see section 2.3.3). A high level example of our fuzzing setup is displayed in Figure 3.1.

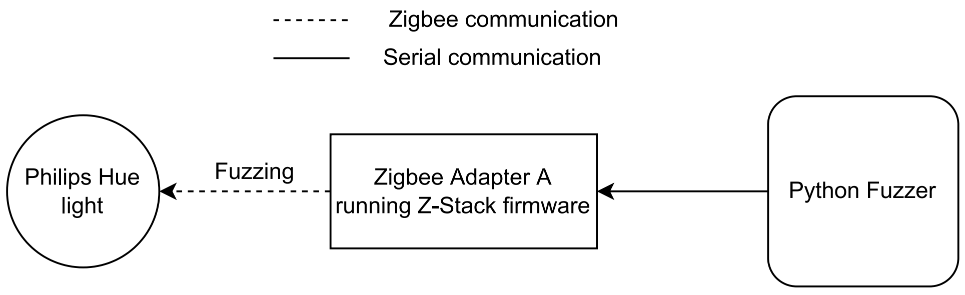


Figure 3.1: High level fuzzing setup

In this part of the bachelor thesis we will perform the actual experiments. A brief reading guide summarizing the key parts of the experiments follows:

- We flash the Z-Stack firmware to Adapter A in section 3.1.
- In section 3.2 we install Zigbee2MQTT and pair our SUT to verify that we have a working setup.
- In section 3.3 we look at sniffing the serial traffic to Adapter A. Using Interceptty we can sniff some interesting data in section 3.3.3.

- In section 3.4 we gain a better understanding of Zigbee messages over the air by sniffing Zigbee traffic. Furthermore, we compare the serial frames to their corresponding Zigbee messages in section 3.4.4.
- Finally, in section 3.5 we will actually fuzz Zigbee messages using Adapter A.

3.1 Flashing Zigbee Adapter A

The goal of this experiment is to have custom firmware on Zigbee Adapter A, with this firmware we should be able to send custom messages. Flashing is quite straight-forward, since we can use the USB connection for flashing.

To flash the adapter, we follow the tutorial[1] on the manufactures' website[2]. This means we first have to download the coordinator firmware[3], since this is the role that we want to play. For this adapter, the Sonoff Zigbee 3.0, the CC1352P2 CC2652P launchpad coordinator firmware is suggested. We can flash this using the automatic upload tool from JelmerT[4]. We can use the following command to flash the firmware.

```
./cc2538-bsl.py -ewv -p /dev/ttyUSB0 --bootloader-sonoff-usb
    ↪ CC1352P2_CC2652P_launchpad_coordinator_20211217.hex
```

This should give the following result, which means a successful flash:

```
Opening port /dev/ttyUSB0, baud 500000
Reading data from ./CC1352P2_CC2652P_launchpad_coordinator_20211217.hex
Firmware file: Intel Hex
Connecting to target...
CC1350 PG2.0 (7x7mm): 352KB Flash, 20KB SRAM, CCFG.BL_CONFIG at 0
    ↪ x00057FD8
Primary IEEE Address: 00:12:4B:00:24:C0:A9:2A
    Performing mass erase
Erasing all main bank flash sectors
    Erase done
Writing 360448 bytes starting at address 0x00000000
Write 104 bytes at 0x00057F980
    Write done
Verifying by comparing CRC32 calculations.
    Verified (match: 0xba5c19c5)
```

[1] https://sonoff.tech/wp-content/uploads/2022/01/SONOFF-Zigbee-3.0-USB-dongle-plus-firmware-flashing-.pdf

[2] https://itead.cc/product/sonoff-zigbee-3-0-usb-dongle-plus/

[3] https://github.com/Koenkk/Z-Stack-firmware/tree/master/coordinator/Z-Stack_3.x.0

[4] https://github.com/JelmerT/cc2538-bsl

3.2 Initial test with Zigbee2MQTT

The goal of this experiment is to test our setup and if all communications are working properly. If this experiments succeeds, we have a working setup to work from. To test our setup, we will be using Zigbee2MQTT(see section 2.3.4).

Hardware setup and data flow

The setup consists of a computer running Zigbee2MQTT, in this computer the Zigbee Adapter A is plugged in. Communication is shown in this diagram.

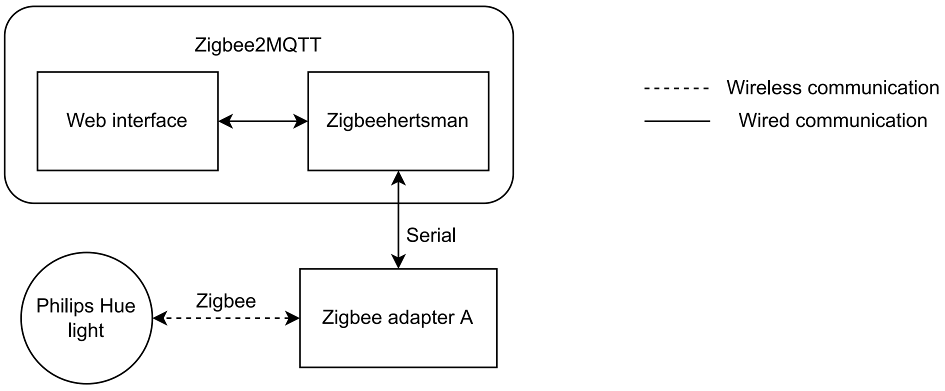


Figure 3.2: Data flow diagram for Zigbee2MQTT

To test if the setup was working, we will set up Zigbee2MQTT. As shown in Figure 3.2, Zigbee2MQTT acts as a bridge to let internet capable devices communicate over MQTT with Zigbee devices.

Setting up software

Zigbee2MQTT requires a folder to work in. The following folder was created for this purpose /home/tom/zigbee2mqtt−data. Inside this folder we will create a configuration file. This file must be named configuration.yaml and contains these lines when starting:

```
permit_join: false
mqtt:
  base_topic: zigbee2mqtt
  server: mqtt://mqtt
serial:
  port: /dev/ttyUSB0
  adapter: auto
frontend:
  port: 8080
```

Installation of the actual program Zigbee2MQTT using Docker seems the simplest. So, let us create the following docker-compose file: (Notice that we also create a simple MQTT server, which is required for internal communication.)

```
version: "3"
services:
  mqtt:
    image: eclipse-mosquitto:2.0
    restart: unless-stopped
    volumes:
      - "./mosquitto-data:/mosquitto"
    ports:
      - "1883:1883"
      - "9001:9001"
    command: "mosquitto -c /mosquitto-no-auth.conf"

  zigbee2mqtt:
    container_name: zigbee2mqtt
    restart: unless-stopped
    image: koenkk/zigbee2mqtt
    volumes:
      - /home/tom/zigbee2mqtt-data:/app/data
      - /run/udev:/run/udev:ro
    ports:
      - 8080:8080
    environment:
      - TZ=Europe/Berlin
    devices:
      - /dev/ttyUSB0:/dev/ttyUSB0
```

After following the tutorial[5], the Zigbee2MQTT server was running on port 8080.

Connecting devices

Adding the Philips Hue light is easy. According to the guidance page[6] we can use the built-in Touchlink[7] functionality. After using Touchlink to factory reset the bulb, it would automatically register itself to Zigbee2MQTT, which means we can control it. As we can see from Figure 3.3, the Philips Hue bulb is added to the user interface and controlling the light is possible.

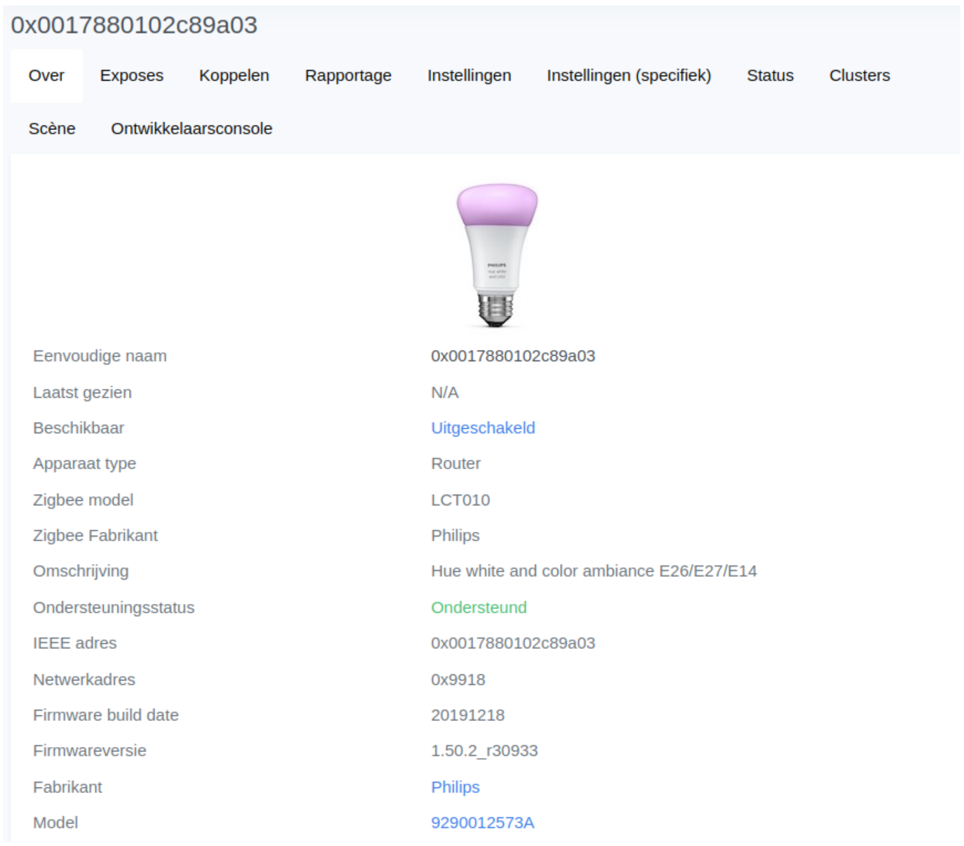
3.3 Sniffing the serial connection to Adapter A

The ultimate goal of this thesis as explained in section 1 is to find a way to control the adapter without using the regular user interface that Zig-

[5]https://www.zigbee2mqtt.io/guide/installation/02_docker.html

[6]https://www.zigbee2mqtt.io/devices/9290012573A.html

[7]https://www.zigbee2mqtt.io/guide/usage/touchlink.html


0x0017880102c89a03

Eenvoudige naam	0x0017880102c89a03
Laatst gezien	N/A
Beschikbaar	Uitgeschakeld
Apparaat type	Router
Zigbee model	LCT010
Zigbee Fabrikant	Philips
Omschrijving	Hue white and color ambiance E26/E27/E14
Ondersteuningsstatus	Ondersteund
IEEE adres	0x0017880102c89a03
Netwerkadres	0x9918
Firmware build date	20191218
Firmwareversie	1.50.2_r30933
Fabrikant	Philips
Model	9290012573A

Figure 3.3: Zigbee2MQTT Philips Hue

bee2MQTT uses so that we can fuzz the connected device. To work towards that, we will in this experiment be looking at the data that flows from Zigbee2MQTT to the adapter. Recall from the communication diagram in Figure 3.2 on page 17 belonging to the previous experiment that the data from Zigbee2MQTT uses a serial communication channel. This serial communication level is the lowest level that we can use to communicate with the adapter. Firstly, we try to use cat in section 3.3.1 to look at this data, but this is not successful. In section 3.3.2 we will use Interceptty to successfully inspect traffic. Other possibilities to control the Zigbee adapter include using the Zigbeehertsman library. For simplicity, we will use the term serial frame as to discuss one transmission/command over the serial communication channel.

3.3.1 Attempt 1: Sniffing serial bus using cat

Beware, this experiment was unsuccessful, please take a look at section 3.3.2 for a proper way to sniff the serial port.

So, it seems like a good start to observe the data that is sent over this serial communication channel. The adapter is mounted on /dev/ttyUSB0. We are going to try to see the communication that happens between Zigbee2MQTT and the adapter, which hopefully gives some insight in how it works.

To do this, we will be using cat to listen to the port

```
cat /dev/ttyUSB0
```

Unfortunately, this only gives gibberish in the terminal and it seems to be interfering with the connection from Zigbee2MQTT as it gives a lot of errors in the log. This means that we can observe two direct problems:

1. The data is (possibly) encoded, so we must decode it.
2. Only reading the data from the serial channel interferes with Zigbee2MQTT, since controlling the light does work a bit, but the state does not update when running cat.

3.3.2 Attempt 2: Sniffing serial bus using Interceptty

This time, we will use a proper program to sniff the traffic going to Zigbee Adapter A. For this, we will use Interceptty[8]. Interceptty basically acts like a Man-in-the-middle on a serial level (see Figure 3.4).

Setup

Installation is explained well in the README file. What we will do is mount the Zigbee adapter on port /dev/ttyUSB0 (as usual). Using Interceptty we will make a virtual mapping to /dev/pts/2. Zigbee2MQTT will now connect to /dev/pts/2. Interceptty will forward all traffic between these ports and also show it in the terminal (or output it to a file).

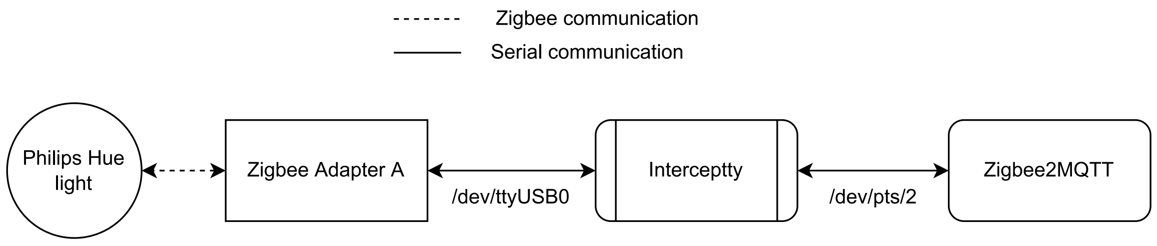


Figure 3.4: Serial port sniffing setup using Interceptty

[8]https://github.com/geoffmeyers/interceptty

Preparation

Before we can do this, we need to prepare a few things:

1. Let Interceptty create a virtual port. This looks like /dev/pts/X. We can find X by simply doing ls /dev/pts/ before Interceptty is running and while it is running to find which port is created. For now, we assume this is port /dev/pts/2.

2. Configure Zigbee2MQTT to use port /dev/pts/2: We do this by setting port: /dev/pts/2 in the Zigbee2MQTT configuration file, see section 3.2.

3. We need to change the Zigbee2MQTT docker-compose file from section 3.2 such that it can see the new port. Adding it as a device does not seem to work, but we can mount the entire virtual ports folder to the container by adding – /dev/pts:/dev/pts to the volumes.

To perform this experiment, we will run Interceptty in the first terminal and start the Zigbee2MQTT docker in another.

Note: this method may sometimes fail to set up a proper connection between Zigbee2MQTT and Adapter A. This can be fixed by configuring Zigbee2MQTT to use the adapter directly at port /dev/ttyUSB0 and then start it. When started, we can start Interceptty and point Zigbee2MQTT to the virtual port. Afterwards, we can restart Zigbee2MQTT.

Results

We start Interceptty using:

```
interceptty /dev/ttyUSB0
```

The output will be in the following format, according to the Interceptty manual (man interceptty).

```
Output lines are in this general format:
< 0x54 (T)
> 0x4b (K)
^ Direction
  ^^^^ Hex code (to real device)
       ^^^ ASCII character (to real device)
        ^^^^ Hex code (from real device)
             ^^^ ASCII character (from real device)
```

When we now start our Zigbee2MQTT Docker container, we see a lot of communicating between the adapter and Zigbee2MQTT. Now, when we turn on the light from within Zigbee2MQTT, we see the following happening.

```
< 0xfe
< 0x0d
< 0x24
< 0x01
< 0x18
< 0x99
< 0x0b
< 0x01
< 0x06
< 0x00
< 0x04
< 0x00
< 0x1e
< 0x03
< 0x01
< 0x05
< 0x01
< 0xb9
```

Note: Interceptty also shows the ASCII character corresponding to a byte, but these are omitted since they provide us with no information. We can now decode this data using the Z-Stack API, see section 2.4.3. Using a simple python program[9] we can convert the serial frame to a more understandable format, as displayed in Figure 3.5.

Command	Data
SOF	0xfe
LEN	0x0d
CMD0	0x24
CMD1	0x01
DstAddr	0x18
DstAddr	0x99
DstEndpoint	0x0b
SrcEndpoint	0x01
ClusterId	0x06
ClusterId	0x00
TransId	0x04
Options	0x00
Radius	0x1e
Len	0x03
Data	0x010501
FCS	0xb9

Figure 3.5: Sniffed serial frame

[9] `https://gitlab.science.ru.nl/trust/z-stack-python/-/blob/main/Interceptty_Scraper.py`

3.3.3 Sniffing more data using Interceptty

In this attempt of sniffing serial communication, we will be sniffing more examples of commands to the light bulb to see which parameters change during normal use. We do that using the same setup as in section 3.3.2. We will send commands to the light bulb and then sniff the data to Zigbee Adapter A, as displayed in Figure 3.4. Recall from Figure 3.5 how a serial frame looks like. Some columns are omitted in Figure A.1 since they hold the same value for each row. Using the Z-stack API (see section 2.4.3) it will be explained why these are excluded from the table in Figure A.1.

- SOF, this is the start of a serial frame, which is always 0xfe
- CMD0 and CMD1 hold the values 0x24 and 0x01, which means an AF_DATA_REQUEST according to section 3.2.1.2 from the Z-stack API.
- DstAddr holds the value 0x1899, which is the beginning of the address of our Philips Hue light bulb (in different endianness), see Figure 3.3.
- DstEndpoint, SrcEndpoint also remain the same, namely 0x0b and 0x01.
- Options, which is 0x00, meaning no special options are used. These options may be interesting while fuzzing in section 3.5
- Radius, which is constantly 0x1e, this refers to the number of hops allowed. Recall (from section 2.1.1) that Zigbee is a mesh protocol.

We are now left with the following fields:

- LEN, this is the total length of the frame.
- ClusterId, 2 bytes, specifying the cluster Id, in our case this is either 0x0600, 0x0800 and 0x0003. A different cluster means a different type of command. For example:
  - 0x0600 is used for on/off commands.
  - 0x0003 is used for Color Control.
  - 0x0800 is used for Brightness control.
- TransId, The transaction sequence number, this seems to be a number that increases each time.
- Len, The length of Data.
- Data, The actual data being sent to the application framework of the Light Bulb.

- FCS, The Frame Check Sequence, this is a XOR of all bytes starting from LEN.

So, using this serial interface we can send Zigbee traffic over the air. In section 3.4.4 we will find out how this traffic translates into Zigbee messages.

3.4 Sniffing Zigbee traffic over the air

In section 3.3 we have sniffed the serial communication between Zigbee Adapter A and Zigbee2MQTT. However, in this experiment, we focus on sniffing the aerial communication between Zigbee Adapter A and the Philips Hue Light bulb (SUT). For a visual representation, see Figure 3.6. For simplicity, we will refer to our intercepted packages as Zigbee messages. This is used to discuss an over-the-air transmission using Zigbee.

Before we actually start fuzzing, we try to do some sniffing on Zigbee level[10]. Our goal is to see actual Zigbee traffic, such that later we can see if we are actually fuzzing. We do this by using a second Zigbee adapter, see 2.3.1. This adapter is connected to a computer running Wireshark[11] (see Figure 3.6).

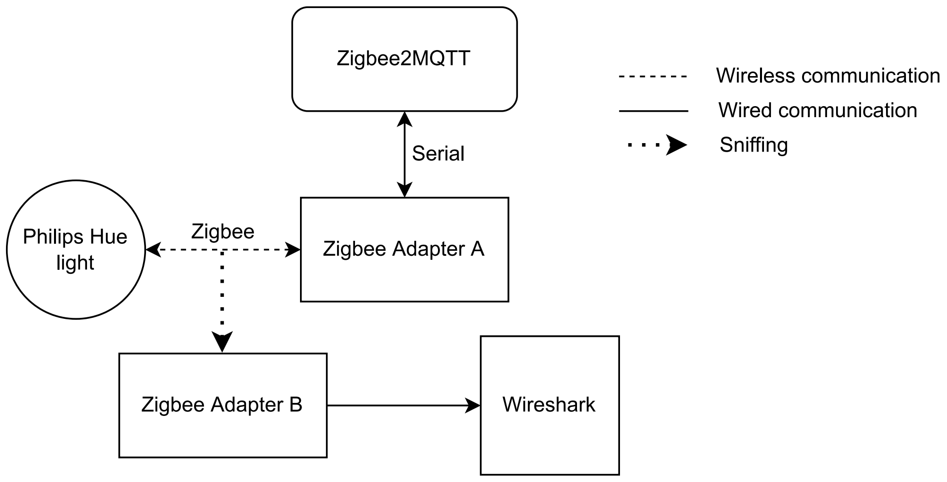


Figure 3.6: Zigbee Sniffing setup: Adapter B eavesdrops on communication between Adapter A and the Philips Hue light.

We install the required firmware on Zigbee Adapter B in section 3.4.1 and install Wireshark on the computer in section 3.4.2. After that, we sniff some basic on/off commands in section 3.4.3 and we determine our options with Zigbee Adapter A in section 3.4.4.

[10] https://www.zigbee2mqtt.io/advanced/zigbee/04_sniff_zigbee_traffic.html#with-cc2531

[11] https://www.wireshark.org/download.html

3.4.1 Flashing Zigbee Adapter B

Firstly, we will download the firmware from Texas Industries[12] (We need PACKET−SNIFFER, not PACKET−SNIFFER−2).

Now, we can unzip the package and convert it to a programmable format:

```
unzip swrc045z.zip −d PACKET−SNIFFER
7z e PACKET−SNIFFER/Setup_SmartRF_Packet_Sniffer_2.18.0.exe bin/general/
    ↪ firmware/sniffer_fw_cc2531.hex
objcopy −−gap−fill 0xFF −−pad−to 0x040000 −I ihex sniffer_fw_cc2531.hex −O
    ↪ binary /packet_sniffer.bin
```

This gives us the packet_sniffer.bin file in our directory.

Unfortunately, flashing this adapter is much more complicated than Adapter B in section 3.1. This is mainly because Adapter A does not require additional hardware to enter the flashing mode. To flash Adapter B, there are multiple options[13]. Since the tutorials are well explained, we will only briefly look at flashing the adapter using a ESP8266 microcontroller[14].

We require CCLoader, installation is trivial:

```
git clone git@github.com:RedBearLab/CCLoader.git
cd CCLoader/SourceCode/Linux
gcc main.c −o CCLoader
```

We will then need to flash our ESP8266 microcontroller with firmware dedicated to flashing.
We need to install the CCLoader/Arduino/CCLoader/CCLoader.ino file. Since we use an ESP8266, we will use the following pin configuration.

```
int DD = 14; //GPIO14=D5 on NodeMCU/WeMos D1 Mini
int DC = 4; //GPIO4=D2 on NodeMCU/WeMos D1 Mini
int RESET = 5; //GPIO5=D1 on NodeMCU/WeMos D1 Mini
int LED = 2; //GPIO2=D4
```

We now connect the ESP8266 to the adapter using the connections shown in Figure 3.7.
When connected, we can start flashing by doing

```
CCLoader/SourceCode/Linux/CCLoader /dev/ttyUSB0 packet_sniffer.bin 1
```

We now see the flashing commences and finishes.

3.4.2 Installing Software

We use Wireshark to inspect packages, Wireshark is commonly available on all platforms and it allows for a Zigbee implementation. Installation

[12]https://www.ti.com/tool/PACKET-SNIFFER

[13]https://www.zigbee2mqtt.io/guide/adapters/flashing/flashing_the_cc2531.html

[14]https://www.zigbee2mqtt.io/guide/adapters/flashing/alternative_flashing_methods.html#via-arduino-uno-esp8266-with-ccloader-3min

ESP8266	CC Pin	CC Name
GND	1	GND
D1	7	RESETn
D2	3	DC
D5	4	DD

Figure 3.7: Wiring Connections

of Wireshark and the Zigbee implementation whsniff[15] is done using the following commands:

```
cd Downloads/
sudo apt-get install -y libusb-1.0-0-dev wireshark
curl -L https://github.com/homewsn/whsniff/archive/v1.3.tar.gz | tar zx
cd whsniff-1.3
make
sudo make install
```

3.4.3 Sniffing basic Zigbee Traffic

In this section we actually sniff some basic Zigbee traffic. Basic in this case means that we send a simple command to turn the Philips Hue light on and off. We do that using a second Zigbee adapter connected to Wireshark. Assuming that we have flashed the adapter (see section 3.4.1) and installed both Wireshark and whsniff (see section 3.4.2).

Setup

Firstly, we create a pipe for Wireshark to connect to:

```
mkdir /tmp/pipes
mkfifo /tmp/pipes/whsniff
```

Now we start whsniff and Wireshark, configured to this pipe:

```
sudo whsniff -c 11 > /tmp/pipes/whsniff &
sudo wireshark -k -i /tmp/pipes/whsniff
```

In the first command, we say `-c 11` to configure the channel to be 11. This is the default for Zigbee2MQTT. We can add our Zigbee keys to Wireshark. The network key can be found in the `configuration.yaml` file (see section 3.2). The Trust Center link key (see section 2.1.3) is (in our case): `5A:69:67:42:65:65:41:6C:6C:69:61:6E:63:65:30:39`. These keys can be added to Wireshark via Edit, Preferences, Protocols, Zigbee, Edit.

[15] https://github.com/homewsn/whsniff

Results

After having done all this, we can see all Zigbee traffic flowing around. This mainly consists of broadcasts every 15 seconds. When we turn the light on and off in sequence, we see the following in Figure 3.8.

Source	Destination	Protocol	Length	Info
0x0000	0x9918	ZigBee HA	48	ZCL OnOff: On, Seq: 29
		IEEE 802.15.4	5	Ack
0x9918	0x0000	ZigBee HA	50	ZCL: Default Response, Seq: 29
		IEEE 802.15.4	5	Ack
0x0000	0x9918	ZigBee HA	48	ZCL OnOff: Off, Seq: 30
		IEEE 802.15.4	5	Ack
0x9918	0x0000	ZigBee HA	50	ZCL: Default Response, Seq: 30
		IEEE 802.15.4	5	Ack

Figure 3.8: Captured Zigbee messages from a Wireshark export

So when we inspect these packages, we see that 0x0000 requests 0x9918 to turn on, on which 0x9918 gives the Acknowledgement. We can see the same for the turn-off command.

3.4.4 Sniffing advanced Zigbee packages

We now need to figure out the following: Consider s to be a serial frame to Adapter A and z to be the Zigbee message resulting from this frame (see Figure 3.9). Since z is encrypted, we will also have z decrypted, which we will call z'. Because encryption is done on the Adapter, we will focus on the decrypted message z'. What is the relation between s and z'? When we know how we can influence z and z' using s, we can figure out our possibilities to fuzz in section 3.5.

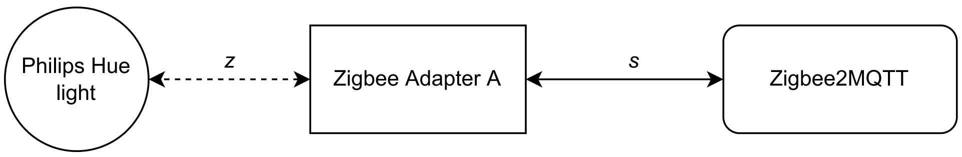


Figure 3.9: Adapter A transforms serial frame s into Zigbee message z.

We do this by combining experiment 3.4.3 and 3.3.3. Since looking at plain bytes can be a little abstract, we will also include the command from Zigbee2MQTT, represented by the variable m. This means we will be sniffing both the serial traffic to Adapter A and the Zigbee traffic over the air from Adapter A to the SUT, whilst also keeping track of the commands being sent by Zigbee2MQTT. See Figure 3.10 for a visual representation of our setup.

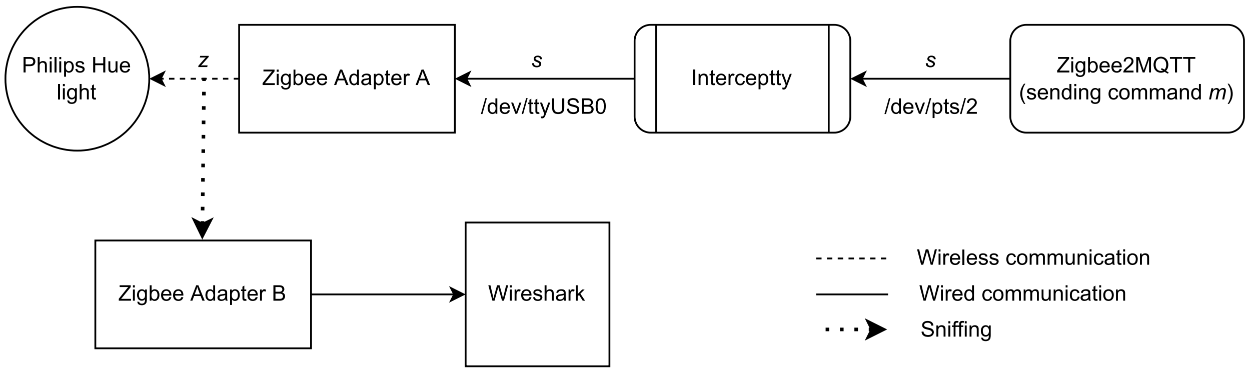


Figure 3.10: Visual representation of experiment 3.4.4, with serial frame s, Zigbee message z and Zigbee2MQTT command m.

Setup

1. We start Zigbee2MQTT with Interceptty fuzzing the serial port, see experiment 3.3.2. The logs from Zigbee2MQTT will be printed to a terminal as well.
2. We start Wireshark connected to Zigbee Adapter B in the same way as in experiment 3.4.3.

We now have a sight on both serial frame s, Zigbee2MQTT log m and Zigbee message z, which we will compare to find differences and similarities. See Figure A.2 for s and Figure A.4 for m. For z it is a bit more complicated, since z is encrypted (see section 2.1.3). Luckily, we know the keys from Zigbee2MQTT and Wireshark can give us the decrypted payload z' (see Figure A.3).

Results

The format of serial frames s and their relation to Zigbee messages z and z' is described in 3.11. More information on the serial frames can be found in section 2.4.3. In Figure 3.12 we see the Zigbee protocol stack and the format of Zigbee messages z and z' is described in Figure 3.13. This information is extracted from sniffed packages in Wireshark and can be found on page 30.

DstAddr	Dst-Endpoint	Src-Endpoint	Cluster	Trans	Options	Radius	Len	Data
z	z'	z'	z'			z		z'

Figure 3.11: Serial frame format of AF_DATA_REQUEST (s). A z or z' in a field means that this field directly occurs in z or z'. DstAddr and Cluster are 2 byte fields, Data has as many bytes as specified in Len. All other fields are 1 byte.

When we put the tables next to each other, there are a few interesting observations when comparing s with z'.

1. Destination endpoint, source endpoint and cluster are both available in s and z'.
2. The data being sent in z' is exactly the same as in s, but without an extra length value.
3. There is a counter in z', which is not in s
4. z' has a 2 byte profile. Which in our case is Home Automation. However, this is not visible in s.

We can also compare s with z in Wireshark, this gives the following observations:

1. The destination address is visible in both s and z. In z, it is in the MAC layer and the Network layer (see Figure 3.13). In both places, they are represented in little-endian. This can be confusing, since we read them in big-endian.
2. When we focus more on the Network layer in z, we see that the radius from s is also visible.

This means there are a few ways to influence z and z' by altering s. However, there are not very many possibilities since the Z-Stack firmware running on our adapter plays a very fundamental role in converting the serial frames to Zigbee messages. In section 3.5.3 we will discuss how to use this to fuzz on a Zigbee level.

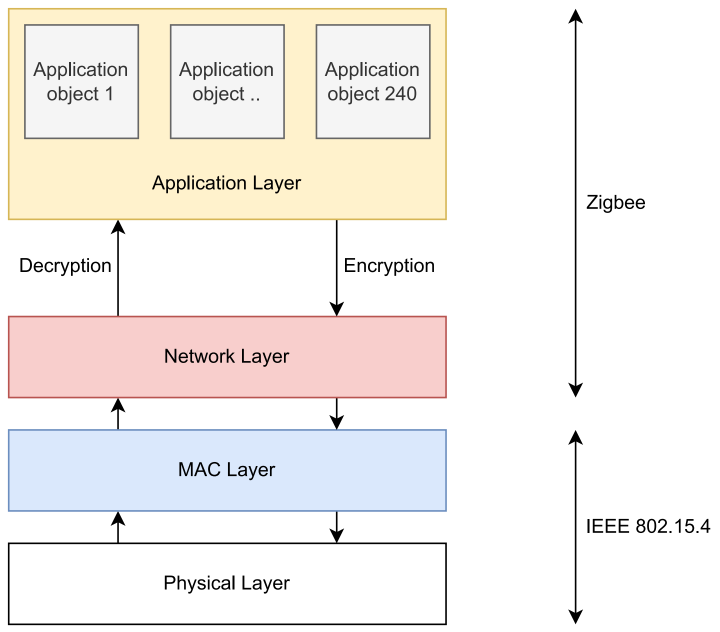


Figure 3.12: Zigbee Network Stack.

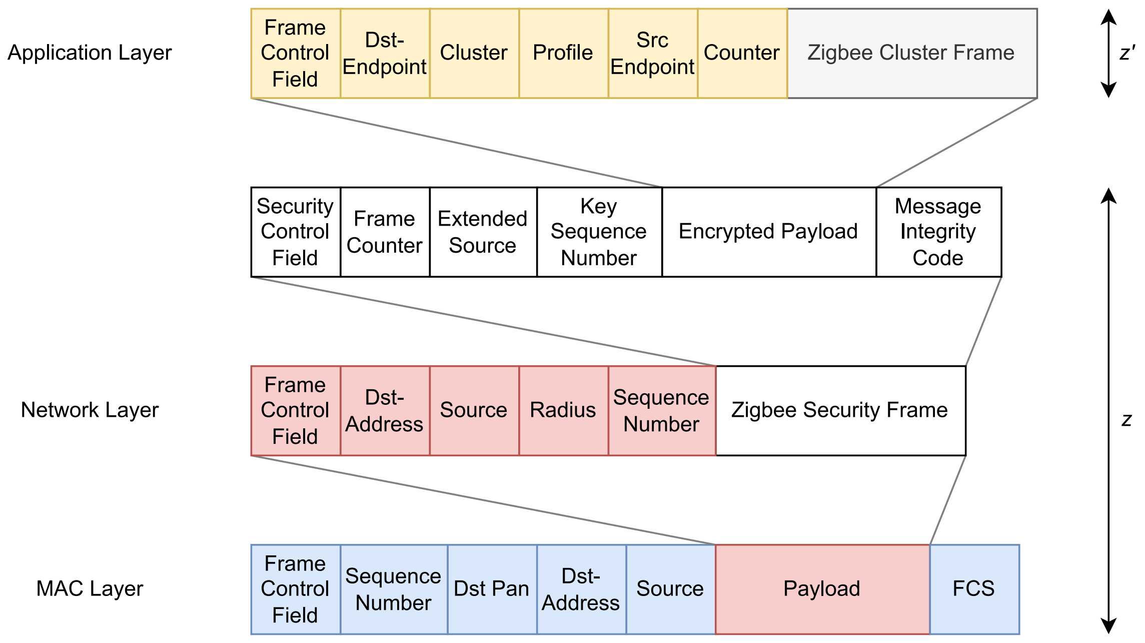


Figure 3.13: Zigbee message z split up into layers.

3.5 Actively fuzzing Zigbee

In this section we perform the actual fuzzing experiments. First, we reason about a crash detective in section 3.5.1. This is a way to determine whether our SUT has crashed. Then, the programming basics and our setup are explained in section 3.5.2. In section 3.5.3 we fuzz a connected device using the command AF_DATA_REQUEST and in section 3.5.4 we fuzz an unpaired device using the command AF_DATA_REQUEST_EXT. Both section 3.5.3 and 3.5.4 first have a subsection regarding the fields to be fuzzed and also a subsection on the results of the fuzzing. Finally, we unsuccessfully try to fuzz using the AF_MSG command in section 3.5.5.

3.5.1 Crash Detective

Foremost, a crash detective is a way for us to determine if the SUT (System under test) has crashed. For a normal computer program running locally, this is typically very trivial, since it will throw an error or stop working. On the contrary, it is difficult to determine whether a piece of software has crashed when you do not have access to the device it is running on. This is a common problem with IoT devices. Research into a generic IoTFuzzer[3] has been done successfully. What basically happens is that they reverse engineer the app corresponding to an IoT device and then fuzz the device via the internet until it gives back unexpected responses. However, our SUT is not directly connected to the internet and we have no physical access to the Philips Hue Light bulb other than its power usage. Nevertheless, remember from experiment 3.4.3 that there is a broadcast frame every 15 seconds (see Figure 3.14). We could use these frames to determine if our SUT has stopped working, since it should send these broadcasts every 15 seconds.

Time	Source	Destination	Protocol	Length
65.222771	0x9918	Broadcast	ZigBee	50
69.637147	0x0000	Broadcast	ZigBee	50
80.303484	0x9918	Broadcast	ZigBee	50
85.425507	0x0000	Broadcast	ZigBee	50

Figure 3.14: Broadcast frames example from Wireshark.

3.5.2 Creating our fuzzer

In this section we discuss what our fuzzer looks like, how features are implemented and why. The basic idea is explained in Figure 3.9. Since the fuzzer uses the Z-stack API, this is the main reason for most choices. The setup used will also be explained.

Fuzzing Program

For simplicity reasons we shall use Python to send the serial frames. Python supports serial communication using the pyserial library:

```
python -m pip install pyserial
```

We use a serial frame to represent one command to the Z-stack API (which runs on Adapter A, see section 2.4). Since we want to mutate bytes easily, we use a list for a serial frame. This list contains all bytes. From 2.4.1 we know that all frames end with a checksum. This can be computed using the following function.

```
def byte_xor(ba1, ba2):
    return bytes([_a ^ _b for _a, _b in zip(ba1, ba2)])

def computeFCS(data):
    FCS = b'\x00'
    for d in data[1:]: # See Z-stack API 2.2.1 (XOR over all but SOF)
        FCS = byte_xor(FCS, d)
    return FCS
```

Some observations have to be made regarding our fuzzer.

1. There are quite a lot of bytes that we can change that influence the Zigbee message. This means there are a lot of possible serial frames that we can send. Imagine that we want to change 3 bytes, this means $(2^8)^3 = 16.777.216$ different possibilities. This costs an absurd amount of time, especially if it is more than 3 bytes. Hence, we fuzz most fields independently of other fields.

2. When we send more than 10 serial frames per second to Zigbee Adapter A, we only see some of them appearing on Wireshark. This means either Adapter A can only send 10 Zigbee messages per second or the sniffer can only sniff 10 Zigbee messages per second. In any case, in order to fuzz thoroughly, we can only send a finite number of Zigbee messages per second. That is why we must consider a timeout between each serial frame.

With this in mind, we use the following functions for sending data, generating random bytes and generating all 1 byte bytes.

```
def sendFrame(f):
    for s in f:
        ser.write(s)
    time.sleep(TimeOut)

def getRandomBytes(size=32) -> []:
    result = []
    for i in range(size):
        result.append(os.urandom(1))
    return result

def getByteList(size=256) -> []:
    result = []
    for i in range(size):
        result.append(i.to_bytes(1, 'big'))
    return result
```

The rest of the code is on GitLab[16].

Setup

Before we start, we first have to start Adapter A and get it in the coordinator mode. Followed by an initial pairing where Adapter A and the SUT get paired. We do this by simply starting Zigbee2MQTT (see section 3.2) and shutting it down. This only needs to be done once, since the devices will be paired until a power reset.

After that, we can use the setup as displayed in Figure 3.15. Firstly, we start our Zigbee sniffer with Wireshark as explained in section 3.4.3. For extra information, we use Interceptty to sniff our serial traffic. We start this using

```
interceptty /dev/ttyUSB0
```

Now we can connect to the virtual port using our Python fuzzer, we can find this port in \dev\pts\ . With this setup, we can finally send our custom messages over Zigbee on a small scale.

Initial results

After sending some messages, the following things were remarkable:

1. Using a different cluster field will result in the Zigbee message being interpreted differently.
2. TransId does not influence the sequence number anywhere.
3. The sequence number in the data is displayed, but nothing seems to be done with it.

[16]https://gitlab.science.ru.nl/trust/z-stack-python/-/blob/main/Serial_Fuzzer.py

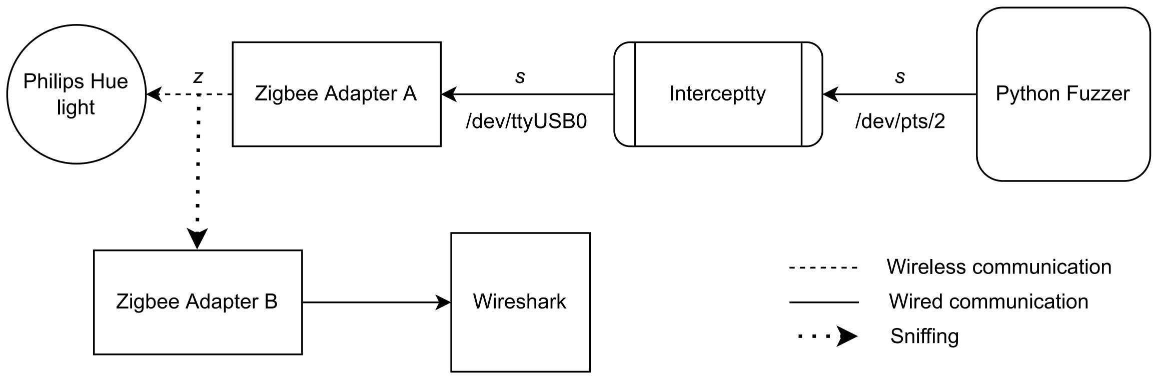


Figure 3.15: Visual representation of the fuzzing setup of the experiments in section 3.5 with serial frame s and Zigbee message z.

3.5.3 Fuzzing the AF_DATA_REQUEST command

In this section we communicate with the Z-Stack firmware on Zigbee Adapter A using the AF_DATA_REQUEST method from the Z-Stack API, see section 2.4.3. We use this command because it is the command used by Zigbee2MQTT to send data, which means that it has a good chance of working. In this scenario, a Zigbee device is paired with a malicious coordinator (see section 2.1.1 for information on coordinators). In our case, the coordinator is Zigbee Adapter A and the Zigbee device is the Philips Hue Light bulb.

In the first part, the fields that we fuzz are explained and numbered. Such that we can easily present the results in the second part. Finally, we present a diagram with the Zigbee messages that show which parts of the network stack we can change in Figure 3.17.

Fields to be fuzzed

In this section we will discuss which fields of the serial frame we alter in our fuzzing attempt. In section 3.4.4 we sniffed serial traffic to Adapter A and Zigbee traffic over the air to find similarities. Using these findings, we will determine which fields of the frame to fuzz. See Figure 3.16 for the fields in a serial frame.

1. We have full control of the Data field which results in the Zigbee Cluster Frame. According to the Z-Stack API, this may have a length of $1 - 128$ bytes. This means it is infeasible to fuzz all possible commands. The second byte is used for the sequence number, we can also fuzz this specifically.

2. The Data field is preceded by the Len field. We shall refer to this as data-length. Remember from section 2.4.1 that we also have a length

value at the beginning of every serial frame (which we call frame-length). However, these values seem unrelated to Zigbee messages. Meaning that probably special nothing makes it onto the air.

3. Destination endpoint, source endpoint and cluster (1 byte, 1 byte and 2 bytes) are also being sent to the application layer

4. The destination address (2 bytes) can also be changed and may influence both the MAC and Network Layer. (See Figure 3.13.

5. Different values for radius influence the Network Layer.

6. The Options parameter is used to pass some extra parameters to Adapter A. So trying different values might be interesting.

7. We tried different values for TransId, also in a weird order. This did not result in our SUT crashing or even ignoring the Zigbee messages.

DstAddr	Dst-Endpoint	Src-Endpoint	Cluster	Trans	Options	Radius	Len	Data
f	*z* *f*	*z* *f*	*z* *f*	*f*	*f*	*z* *f*	*f*	*z* *f*

Figure 3.16: Serial frame of AF_DATA_REQUEST. A z means that changing this field directly influences the Zigbee message that results from this frame. f means that we fuzz this field.

Results

Because of time complexity, most fields were fuzzed separately. In total, around 70000 Zigbee messages were generated which took around 2 hours. The Philips Hue light bulb did not crash during these fuzzing attempts. In Figure 3.16 the fields are displayed that have been fuzzed and in Figure 3.17 an example Zigbee message is displayed with only the fields that can be influenced by us. Of these, only the destination addresses can be partially controlled.

1. We have total control of the Zigbee Cluster Frame in Zigbee message z by using the data field in s. The SUT responds to every combination of data with an acknowledgement. Changing the sequence number does not do anything special.

2. The data-length value seems to indicate which of the following bytes need to be interpreted. The same holds for the serial-length value, we see that all bytes after this length will be ignored by Adapter A.

However, we can get a very large Zigbee message with random data when we set the data-length value to be very high (0xf0/127) and not send those 127 bytes. What possibly happens is that we are reading the memory of Zigbee Adapter A.

3. For destination endpoint, only values of 0x0b and 0xff are fully accepted by the SUT. This means that it will handle those commands. Other values of destination endpoint will result in the SUT only sending an acknowledgement.

   For the source endpoint, only some values result in a Zigbee message being sent. These values are $(1-6, 8, 10-13, 47, 110, 242)$. All these values seem to correspond to a different profile (Such as Home Automation and Telecom Automation). The SUT seems to respond only to the Home Automation and ZZL profile.

   When we change the cluster, we see that the Zigbee Cluster Frame is interpreted differently.

4. When a known destination address is sent to Adapter A, it sends a Zigbee message to that address. However, if the address is unknown to Adapter A, it sends a route request. Furthermore, 0xFF 0xFF is the broadcast address and works as expected.

5. Changing the radius has no effect on our setup, since we only have 2 Zigbee devices. This means messages do not have to travel across multiple routers.

6. Trying all 255 different combinations for options in s does not crash our SUT unfortunately. Options is a parameter for AF_DATA_REQUEST and does not directly relate to a field in z.

7. Different values for TransId do not result in anything out of the ordinary. Moreover, a strictly decreasing sequence does not do anything special.

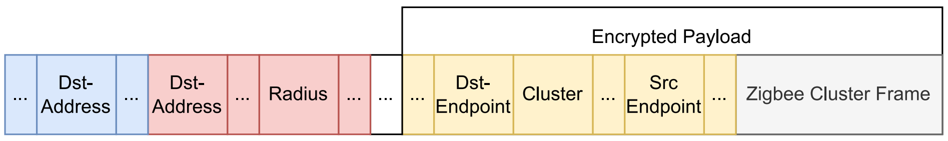


Figure 3.17: Fields in a Zigbee message that can be changed by using the AF_DATA_REQUEST command.(All fields with ... are generated by the Z-Stack firmware.) See figures 3.12 and 3.13 for more information.

3.5.4 Fuzzing the AF_DATA_REQUEST_EXT command

In this section, we try to fuzz a device that is currently paired to another coordinator. This means that our coordinator (Zigbee Adapter A) does not know the address or security keys. This is a very common real life scenario, since all Zigbee devices in domestic use should be paired to a coordinator in order to function properly.

For this experiment, we use a second Philips Hue light bulb that is still connected to the original Philips Zigbee Bridge. We first get the full address using Touchlink and Zigbee2MQTT. After that, we try to fuzz it using AF_DATA_REQUEST_EXT from section 2.4.4.

Getting the Zigbee address and channel

In order to send messages to a light bulb we naturally require its address. However, we also need the channel and PAN on which it is working. We can retrieve the Zigbee address and channel using the Touchlink functionality that we also used in section 3.2. To do this, we start Zigbee2MQTT and put Adapter A in close proximity to our light bulb. We then start the Touchlink scan and find our Philips Hue Light Bulb. Resulting in the address 0x0017880100dcc2b3 on channel 15. We also change our channel to 15 in Zigbee2MQTT by going to "Settings ⟶ Advanced ⟶ Channel". Removal of the coordinator_backup.json file in the Zigbee2MQTT directory is also necessary. Next, we need to configure our sniffer to sniff on channel 15. This means we have to pass the argument –c 15 (see section 3.4.3). When we now sniff traffic, we see broadcast messages from our new light bulb. In which the Destination PAN is visible, which is 0x7cd7. We also see that the short address is 0x0003.

Preparing a AF_DATA_REQUEST_EXT frame

In section 2.4.4 we introduce the AF_DATA_REQUEST_EXT command. This command requires some preparation. For starters, we need to set the DstAddrMode. We collected the address and PanId in the previous section, so we can fill these in (using the proper endianness). Furthermore, we can also use the broadcast addresses, which means setting all bits to 1. The other parameters of AF_DATA_REQUEST_EXT were also used in section 3.5.3. We had no success there tampering with these parameters. Hence, we will focus on different parameters. DstEndpoint and SrcEndpoint are an exception because they may be interesting. This results in us reusing the values gathered in experiment 3.3.2 (see Figure 3.5). This means:

- Cluster: 0x0600
- Options: 0x00

- Radius: 0x1e
- Trans: 0x04
- Len: 0x03
- Data: 0x010002.

Fields to be fuzzed

In this section we discuss which fields of the serial frame we use to fuzz. See Figure 3.18 for a serial frame of AF_DATA_REQUEST_EXT.

1. We try the address of the SUT and the broadcast address in both the short and long notation (see section 2.4.4).
2. We try different values for SrcEndpoint and DstEndpoint.
3. We try different values for PanId, including the broadcast pan (0xffff) and other interesting PanIds[1].

DstAddr-Mode	DstAddr	Dst-Endpoint	PanId	Src-Endpoint	Cluster	Trans	Options	Radius	Len	Data
f	*f*	*f*	*f*	*f*						

Figure 3.18: Serial frame of AF_DATA_REQUEST_EXT. An f means that we fuzz this field.

Results

When sending a serial frame to Adapter A, there are 3 possible outcomes. Either the Z-Stack firmware does not know what to do with this serial frame and does nothing, or it is transformed into a Zigbee message to which our SUT can either respond or not.

1. Zigbee messages with the short notation only get transmitted when we use the broadcast address. This is because the short address (0x0003) is unknown to Adapter A. However, we cannot influence the PanId in that case.

   The long notation in combination with the DstEndpoint set to 0xfe results in Zigbee messages being transmitted with control over the PanId. If we send this to the exact address, we get an acknowledgement back.

2. We can change the SrcEndpoint and DstEndpoint fields in any Zigbee message.

3. We have control over the PanId, but we do not get any responses from the SUT when we use the wrong PanId.

The Zigbee message resulting from the AF_DATA_REQUEST_EXT serial command is shown in Figure 3.19. Because we are sending to an unknown device, we do not know the security keys and thus we are sending the message in plain text.

As stated before, we can get a reply (in the form of an acknowledgement) from our SUT if we do the following:

- We set the correct address in 8 byte length.
- DstEndpoint is set to 0xfe
- Use the correct PAN (0x7cd7) or the broadcast PAN (0xffff)

Note that we can only get an acknowledgement. In other words, the SUT does not actively respond by, for example, turning off. However, since we can trigger a reaction without setting up a connection, we may possibly perform a DDOS attack (See section 5.4).

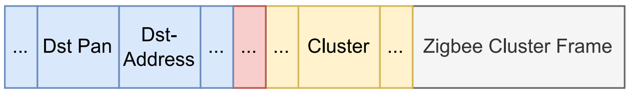


Figure 3.19: Fields in a not encrypted Zigbee message that can be changed by using the AF_DATA_REQUEST_EXT command. (All fields with ... are generated by the Z-Stack firmware.) See figures 3.12 and 3.13 for more information.

3.5.5 Brief experiment: Fuzzing the APP_MSG command

In this experiment we try to fuzz Zigbee messages using the APP_MSG command (see figure 3.20 and section 3.3.1.1 of the Z-Stack API). Unfortunately, sending this serial frame with valid data does not result in a Zigbee message being transmitted. So this command does not seem suitable for use with fuzzing.

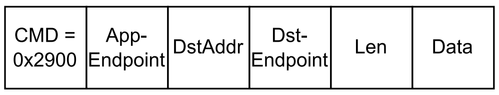


Figure 3.20: Serial frame of the APP_MSG command

Chapter 4

Related Work

Research has been done in attacking Zigbee using XBee[1] devices[22]. In their experiment, the XBee devices occupied all important roles: (Zigbee Coordinator, target device and attacker). These XBee devices were used in combination with microcontrollers or microprocessors such as an Arduino or Raspberry Pi to set up an attack. They decided to only target these XBee devices. This is the main difference when comparing their research with our experiment. Since we targeted consumer grade hardware, such as the Philips Hue Light bulbs that anyone can easily buy.

Their result was that they could send remote AT Commands[2] to nodes in a connected network not using encryption. With these AT Commands they could reconfigure a device, for example to let it join a malicious network.

Research into Fuzzing Zigbee has been done using a Finite State Machine[4]. The FSM here is used to make the fuzzing attacks more targeted on a specific state a Zigbee device is in. Furthermore, it is also being used to monitor if the devices are in the correct state, so as a method of crash detection.

[1] `https://en.wikipedia.org/wiki/XBee`

[2] `https://en.wikipedia.org/wiki/Hayes_command_set`

Chapter 5

Future Work

5.1 Custom Zigbee Firmware

It is possible to program CC2652P chip-sets. Like what we did in section 2.1.1. An interesting extension to this would be to program the CC2652P in such as way that it does not handle any part of the network stack. There are more Zigbee chipsets that support custom firmware and easy flashing, such as the XT-ZB1[1]. With fully custom firmware, fuzzing possibilities would greatly increase.

5.2 Touchlink

In experiment 3.2 we make an initial connection to the Philips Hue light bulb. We use the Touchlink[2] functionality to reset the light bulb. Touchlink is present in most chips made by Texas Industries. Touchlink has already been shown to be a vulnerability[14]. However, Touchlink only works in a proximity of around 10 centimetres, would it be possible to amplify the Touchlink signal in a similar way that key-less cars are hacked?

5.3 Fuzzing as a method for testing

Fuzzing is primarily used as a method for security testing. However, could it also be used for regular testing purposes by developers?

5.4 DDOS attack on Zigbee

In section 3.5.4 we managed to get a reply from an unpaired device. We did this using the AF_DATA_REQUEST_EXT command from the Z-Stack API.

[1]`https://www.cnx-software.com/2021/12/27/xt-zb1-zigbee-ble-devkit-bl702-risc-v-module/`
[2]`https://www.zigbee2mqtt.io/guide/usage/touchlink.html`

Since we got an acknowledgement, we know that we are communicating with this device and we may perform a DDOS attack.

5.5 Other Z-Stack Commands

The Z-Stack API has more commands that lead to a Zigbee message being sent over the air. For example, the ZDO_MGMT_PERMIT_JOIN_RSP command sends a Zigbee message that initiates a pairing operation. There are probably around 10 more commands that can be fuzzed. Setting up such a command only takes around half an hour, whilst the fuzzing process can take much longer.

Chapter 6

Conclusions

We have flashed the Z-Stack firmware to the Sonoff Zigbee Dongle (see section 3.1). This was very straightforward since flashing can be done over USB. This Z-Stack firmware is the piece of software that takes a serial frame from our Python fuzzer[1] and transforms it into a Zigbee message over the air. The Z-Stack firmware is responsible for most fields of any Zigbee message and in particular most of the MAC Layer and the Network Layer of the Zigbee Network Stack (see Figure 3.12). This means that we can not fuzz most fields of these layers and that we are thus limited in our fuzzing possibilities.

Flashing a CC2531 USB-Dongle for sniffing purposes turned out to be considerably more difficult than the Sonoff Zigbee Dongle (see section 3.4.1). This is mainly because the CC2531 requires an external flasher, even though it is also connected via USB.

The addresses in both the serial frames and Zigbee messages are in little-endian, whereas the way we read these addresses is in big-endian. This can be confusing sometimes (see section 3.4.3).

We have discussed that we can only fuzz around 10 Zigbee messages per second (see section 3.5.2). This is either because the Sonoff Zigbee Dongle can only send around 10 Zigbee messages per second or the CC2531 USB-Dongle can only sniff 10 Zigbee messages per second. In any case, this causes quite a bottleneck when fuzzing on a large scale.

We have successfully used the Z-Stack firmware to fuzz Zigbee messages (see section 3.5.3). Among the fuzzed fields were fields of the Application Layer, such as the Zigbee Cluster, Destination endpoint and the Source endpoint (see Figure 3.17). However, fuzzing on these fields did not result in the Philips Hue Light crashing.

We were able to get a response from a Zigbee device that was not paired to our Sonoff Zigbee Dongle (see section 3.5.4). This creates the possibility for a DDOS attack.

[1] `https://gitlab.science.ru.nl/trust/z-stack-python/-/blob/main/Serial_Fuzzer.py`

Bibliography

[1] Atmel. *Atmel AT14615: ZigBee Broadcast Message Handling and Implementation in BitCloud SDK*. Atmel Corporation, August 2016.

[2] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. Directed greybox fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, page 2329–2344, New York, NY, USA, 2017. Association for Computing Machinery.

[3] Jiongyi Chen, Wenrui Diao, Qingchuan Zhao, Chaoshun Zuo, Zhiqiang Lin, XiaoFeng Wang, Wing Cheong Lau, Menghan Sun, Ronghai Yang, and Kehuan Zhang. Iotfuzzer: Discovering memory corruptions in iot through app-based fuzzing. In *NDSS*, 2018.

[4] Baojiang Cui, Shurui Liang, Shilei Chen, Bing Zhao, and Xiaobing Liang. A novel fuzzing method for zigbee based on finite state machine. *International Journal of Distributed Sensor Networks*, 10(1):762891, 2014.

[5] Sinem Coleri Ergen. Zigbee/IEEE 802.15. 4 summary. *UC Berkeley, September*, 10(17):11, 2004.

[6] Patrice Godefroid. Fuzzing: Hack, art, and science. *Communications of the ACM*, 63(2):70–76, 2020.

[7] Patrice Godefroid, Adam Kiezun, and Michael Y. Levin. Grammar-based whitebox fuzzing. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, page 206–215, New York, NY, USA, 2008. Association for Computing Machinery.

[8] Michael Hooper, Yifan Tian, Runxuan Zhou, Bin Cao, Adrian P. Lauf, Lanier Watkins, William H. Robinson, and Wlajimir Alexis. Securing commercial wifi-based uavs from common security attacks. In *MILCOM 2016 - 2016 IEEE Military Communications Conference*, pages 1213–1218, 2016.

[9] British Computer Society Specialist Interest Group in Software Testing. *BCS Software Component Testing Standard.* BCS SIGIST, April 1997.

[10] Sewook Jung, Alexander Chang, and Mario Gerla. Comparisons of zigbee personal area network (pan) interconnection methods. In *2007 4th International Symposium on Wireless Communication Systems*, pages 337–341, 2007.

[11] Koen Kanters. Zigbee2MQTT. `https://www.zigbee2mqtt.io/`. (Accessed on 2022-05-29).

[12] Hongwei Li, Zhongning Jia, and Xiaofeng Xue. Application and analysis of zigbee security services specification. In *2010 Second International Conference on Networks Security, Wireless Communications and Trusted Computing*, volume 2, pages 494–497, April 2010.

[13] Hongliang Liang, Xiaoxiao Pei, Xiaodong Jia, Wuwei Shen, and Jian Zhang. Fuzzing: State of the art. *IEEE Transactions on Reliability*, 67(3):1199–1218, 2018.

[14] Philipp Morgner, Stephan Mattejat, Zinaida Benenson, Christian Müller, and Frederik Armknecht. Insecure to the touch: Attacking zigbee 3.0 via touchlink commissioning. In *Proceedings of the 10th ACM Conference on Security and Privacy in Wireless and Mobile Networks*, WiSec '17, page 230–240, New York, NY, USA, 2017. Association for Computing Machinery.

[15] Srinivas Nidhra and Jagruthi Dondeti. Black box and white box testing techniques-a literature review. *International Journal of Embedded Systems and Applications (IJESA)*, 2(2):29–50, 2012.

[16] Olayemi Olawumi, Keijo Haataja, Mikko Asikainen, Niko Vidgren, and Pekka Toivanen. Three practical attacks against zigbee security: Attack scenario definitions, practical experiments, countermeasures, and lessons learned. In *2014 14th International Conference on Hybrid Intelligent Systems*, pages 199–206, Dec 2014.

[17] C. Muthu Ramya, M Shanmugaraj, and R Prabakaran. Study on zigbee technology. In *2011 3rd International Conference on Electronics Computer Technology*, volume 6, pages 297–301, 2011.

[18] Mengfei Ren, Xiaolei Ren, Huadong Feng, Jiang Ming, and Yu Lei. Z-fuzzer: Device-agnostic fuzzing of zigbee protocol implementation. In *Proceedings of the 14th ACM Conference on Security and Privacy in Wireless and Mobile Networks*, WiSec '21, page 347–358, New York, NY, USA, 2021. Association for Computing Machinery.

[19] Shahnaz Saleem, Sana Ullah, and Kyung Sup Kwak. A study of ieee 802.15.4 security framework for wireless body area networks. *Sensors (Basel, Switzerland)*, 11(2):1383–1395, 2011. 22319358[pmid].

[20] Michael Sutton, Adam Greene, and Pedram Amini. *Fuzzing: brute force vulnerability discovery.* Pearson Education, 2007.

[21] Inc. Texas Instruments. *Z-Stack Monitor and Test API (Revision 1.178)*, 2020.

[22] Ivan Vaccari, Enrico Cambiaso, and Maurizio Aiello. Remotely exploiting at command attacks on zigbee networks. *Security and Communication Networks*, 2017, 2017.

[23] Denny Vrandečić and Aldo Gangemi. Unit tests for ontologies. In Robert Meersman, Zahir Tari, and Pilar Herrero, editors, *On the Move to Meaningful Internet Systems 2006: OTM 2006 Workshops*, pages 1012–1020, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[24] Chonggang Wang, Kazem Sohraby, Rittwik Jana, Lusheng Ji, and Mahmoud Daneshmand. Voice communications over zigbee networks. *IEEE Communications Magazine*, 46(1):121–127, 2008.

[25] Maverick Woo, Sang Kil Cha, Samantha Gottlieb, and David Brumley. Scheduling black-box mutational fuzzing. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, CCS '13, page 511–522, New York, NY, USA, 2013. Association for Computing Machinery.

[26] Min Zhou and Zhang long Nie. Analysis and design of zigbee mac layers protocol. In *2010 International Conference on Future Information Technology and Management Engineering*, volume 2, pages 211–215, 2010.

Appendix A

Appendix

A.1 Results for experiment 3.3.3

See Figure A.1.

A.2 Results for experiment 3.4.4

See figures A.2, A.3 and A.4.

Command	**Data**					
Command	LEN	ClusterId	TransId	Len	Data	FCS
Off	0xd	0x6 0x0	0x14	0x3	0x1 0x15 0x0	0xb8
On	0x10	0x8 0x0	0x15	0x6	0x1 0x16 0x4 0x54 0x0 0x0	0xfc
Brightness 50	0x10	0x8 0x0	0x16	0x6	0x1 0x17 0x4 0x32 0x0 0x0	0x98
Brightness 100	0x10	0x8 0x0	0x17	0x6	0x1 0x18 0x4 0x64 0x0 0x0	0xc0
Brightness 254	0x10	0x8 0x0	0x18	0x6	0x1 0x19 0x4 0x64 0x0 0x0	0xce
Color Temp 0	0x10	0x8 0x0	0x19	0x6	0x1 0x1a 0x4 0xfe 0x0 0x0	0x56
Color Temp 1	0x11	0x0 0x3	0x1a	0x7	0x1 0x1b 0xa 0x99 0x0 0x0 0x0	0x36
Color Temp 2	0x11	0x0 0x3	0x1b	0x7	0x1 0x1c 0xa 0xfa 0x0 0x0 0x0	0x53
Color Temp 3	0x11	0x0 0x3	0x1c	0x7	0x1 0x1d 0xa 0x72 0x1 0x0 0x0	0xdc
Color Temp 4	0x11	0x0 0x3	0x1d	0x7	0x1 0x1e 0xa 0xc6 0x1 0x0 0x0	0x6a
Red	0x11	0x0 0x3	0x1e	0x7	0x1 0x1f 0xa 0xf4 0x1 0x0 0x0	0x5a
Green	0x13	0x0 0x3	0x1f	0x9	0x1 0x20 0x7 0x79 0xa1 0xb5 0x52 0x0 0x0	0xaf
Yellow	0x13	0x0 0x3	0x20	0x9	0x1 0x21 0x7 0xb6 0x7d 0xb6 0x6d 0x0 0x0	0xbe
Brown	0x13	0x0 0x3	0x21	0x9	0x1 0x22 0x7 0x18 0x69 0x4e 0x7d 0x0 0x0	0xee
Blue	0x13	0x0 0x3	0x22	0x9	0x1 0x23 0x7 0xda 0x4b 0xb4 0x97 0x0 0x0	0x1c
Purple	0x13	0x0 0x3	0x23	0x9	0x1 0x24 0x7 0x2d 0x26 0xb8 0xf 0x0 0x0	0x14
Off	0xd	0x6 0x0	0x25	0x3	0x1 0x26 0x0	0xba

Figure A.1: Sniffed serial data with their commands

SOF	LEN	CMD0	CMD1	DstAddr	DstEP	SrcEP	ClusterId	TransId	Options	Radius	Len	Data	FCS
0xfe	0x0d	0x24	0x1	0x1899	0xb	0x1	0x0600	0x2	0x0	0x1e	0x3	0x010301	0xb9
0xfe	0x11	0x24	0x1	0x1899	0xb	0x1	0x0003	0x3	0x0	0x1e	0x7	01 04 0a f4 01 00 00	0x5c
0xfe	0x10	0x24	0x1	0x1899	0xb	0x1	0x0800	0x4	0x0	0x1e	0x6	01 05 04 55 00 00	0xff
0xfe	0x10	0x24	0x1	0x1899	0xb	0x1	0x0800	0x5	0x0	0x1e	0x6	01 06 04 a0 00 00	0x08
0xfe	0x0f	0x24	0x1	0x1899	0xb	0x1	0x0300	0x6	0x0	0x1e	0x5	01 07 40 00 00	0xf9
0xfe	0x0f	0x24	0x1	0x1899	0xb	0x1	0x0300	0x7	0x0	0x1e	0x5	01 08 40 fe 00	0x09
0xfe	0x13	0x24	0x1	0x1899	0xb	0x1	0x0003	0x8	0x0	0x1e	0x9	0x01090779a1b5520000	0x91
0xfe	0x13	0x24	0x1	0x1899	0xb	0x1	0x0003	0x9	0x0	0x1e	0x9	0x010a0718694e7d0000	0xee
0xfe	0x13	0x24	0x1	0x1899	0xb	0x1	0x0003	0xa	0x0	0x1e	0x9	0x010b072d26b80f0000	0x12
0xfe	0x10	0x24	0x1	0x1899	0xb	0x1	0x0800	0xb	0x0	0x1e	0x6	01 0c 04 fe 00 00	0x52
0xfe	0x11	0x24	0x1	0x1899	0xb	0x1	0x0003	0xc	0x0	0x1e	0x7	01 0d 0a 99 00 00 00	0x36
0xfe	0x12	0x24	0x1	0x1899	0xb	0x1	0x0003	0xd	0x0	0x1e	0x8	0x100e02104021c601	0x0e

Figure A.2: Captured Serial frames *s*

DstEP = Destination Endpoint
SrcEP = Source Endpoint

Frame Control Field	Destination Endpoint	Cluster	Profile	Source endpoint	Counter	Zigbee Cluster Library frame
0x00	0x0b	0x6 0x0	0x04 0x01	0x01	0x59	01 03 01
0x00	0x0b	0x0 0x3	0x04 0x01	0x01	0x5a	01 04 0a f4 01 00 00
0x00	0x0b	0x8 0x0	0x04 0x01	0x01	0x5b	01 05 04 55 00 00
0x00	0x0b	0x8 0x0	0x04 0x01	0x01	0x5c	01 06 04 a0 00 00
0x00	0x0b	0x3 0x0	0x04 0x01	0x01	0x5d	01 07 40 00 00
0x00	0x0b	0x3 0x0	0x04 0x01	0x01	0x5e	01 08 40 fe 00
0x00	0x0b	0x0 0x3	0x04 0x01	0x01	0x5f	01 09 07 79 a1 b5 52 00 00
0x00	0x0b	0x0 0x3	0x04 0x01	0x01	0x60	01 0a 07 18 69 4e 7d 00 00
0x00	0x0b	0x0 0x3	0x04 0x01	0x01	0x61	01 0b 07 2d 26 b8 0f 00 00
0x00	0x0b	0x8 0x0	0x04 0x01	0x01	0x62	01 0c 04 fe 00 00
0x00	0x0b	0x0 0x3	0x04 0x01	0x01	0x63	01 0d 0a 99 00 00 00
0x00	0x0b	0x0 0x3	0x04 0x01	0x01	0x64	10 0e 02 10 40 21 c6 01

Figure A.3: Captured Decrypted Zigbee packets z'

```
  zigbee2mqtt | Zigbee2MQTT:info 2022−05−02 17:37:52: MQTT publish: topic '
    ↪ zigbee2mqtt/0x0017880102c89a03', payload '{"brightness":254,"color":{"hue
    ↪ ":66,"saturation":88,"x":0.4105263157894737,"y":0.48947368421052634},"
    ↪ color_mode":"xy","color_temp":252,"color_temp_startup":153,"linkquality
    ↪ ":94,"state":"ON"}'
zigbee2mqtt | Zigbee2MQTT:info 2022−05−02 17:37:55: MQTT publish: topic '
    ↪ zigbee2mqtt/0x0017880102c89a03', payload '{"brightness":254,"color":{"hue
    ↪ ":25,"saturation":95,"x":0.5267,"y":0.4133},"color_mode":"color_temp","
    ↪ color_temp":500,"color_temp_startup":153,"linkquality":94,"state":"ON"}'
zigbee2mqtt | Zigbee2MQTT:info 2022−05−02 17:37:58: MQTT publish: topic '
    ↪ zigbee2mqtt/0x0017880102c89a03', payload '{"brightness":85,"color":{"hue
    ↪ ":25,"saturation":95,"x":0.5267,"y":0.4133},"color_mode":"color_temp","
    ↪ color_temp":500,"color_temp_startup":153,"linkquality":94,"state":"ON"}'
zigbee2mqtt | Zigbee2MQTT:info 2022−05−02 17:38:00: MQTT publish: topic '
    ↪ zigbee2mqtt/0x0017880102c89a03', payload '{"brightness":160,"color":{"hue
    ↪ ":25,"saturation":95,"x":0.5267,"y":0.4133},"color_mode":"color_temp","
    ↪ color_temp":500,"color_temp_startup":153,"linkquality":102,"state":"ON"}'
zigbee2mqtt | Zigbee2MQTT:info 2022−05−02 17:38:19: MQTT publish: topic '
    ↪ zigbee2mqtt/0x0017880102c89a03', payload '{"brightness":160,"color":{"hue
    ↪ ":3,"saturation":96,"x":0.6307692307692307,"y":0.3230769230769231},"
    ↪ color_mode":"xy","color_temp":319,"color_temp_startup":153,"linkquality":94,"
    ↪ state":"ON"}'
zigbee2mqtt | Zigbee2MQTT:info 2022−05−02 17:38:22: MQTT publish: topic '
    ↪ zigbee2mqtt/0x0017880102c89a03', payload '{"brightness":160,"color":{"hue
    ↪ ":66,"saturation":88,"x":0.4105263157894737,"y":0.48947368421052634},"
    ↪ color_mode":"xy","color_temp":252,"color_temp_startup":153,"linkquality":98,"
    ↪ state":"ON"}'
zigbee2mqtt | Zigbee2MQTT:info 2022−05−02 17:38:23: MQTT publish: topic '
    ↪ zigbee2mqtt/0x0017880102c89a03', payload '{"brightness":160,"color":{"hue
    ↪ ":240,"saturation":97,"x":0.14912280701754385,"y":0.06140350877192982},"
    ↪ color_mode":"xy","color_temp":500,"color_temp_startup":153,"linkquality":91,"
    ↪ state":"ON"}'
zigbee2mqtt | Zigbee2MQTT:info 2022−05−02 17:38:27: MQTT publish: topic '
    ↪ zigbee2mqtt/0x0017880102c89a03', payload '{"brightness":254,"color":{"hue
    ↪ ":240,"saturation":97,"x":0.14912280701754385,"y":0.06140350877192982},"
    ↪ color_mode":"xy","color_temp":500,"color_temp_startup":153,"linkquality":98,"
    ↪ state":"ON"}'
zigbee2mqtt | Zigbee2MQTT:info 2022−05−02 17:38:32: MQTT publish: topic '
    ↪ zigbee2mqtt/0x0017880102c89a03', payload '{"brightness":254,"color":{"hue
    ↪ ":212,"saturation":10,"x":0.3131,"y":0.3232},"color_mode":"color_temp","
    ↪ color_temp":153,"color_temp_startup":153,"linkquality":91,"state":"ON"}'
zigbee2mqtt | Zigbee2MQTT:info 2022−05−02 17:38:38: MQTT publish: topic '
    ↪ zigbee2mqtt/0x0017880102c89a03', payload '{"brightness":254,"color":{"hue
    ↪ ":212,"saturation":10,"x":0.3131,"y":0.3232},"color_mode":"color_temp","
    ↪ color_temp":153,"color_temp_startup":454,"linkquality":91,"state":"ON"}'
```

Figure A.4: Zigbee2MQTT logs frames m