BLE Exploitation
Unfit Story of Fitness Trackers

Yogesh Ojha

#whoami

Yogesh Ojha

- From Nepal 🇳🇵
- Cyber Security Analyst @ TCS Cyber Security Unit, India
- IOT & Mobile Application Security
- Machine Learning enthusiast
- Love to Build and Play with Robots and sometimes break them too ;)

Expectations

You can expect:

- Basic Overview of Bluetooth Low Energy
- Bluetooth Classic vs Bluetooth Low Energy
- BLE Stack
- BLE MiTM/Sniffing BLE Packets
- Reverse Engineering the Mobile Applications of Fitness trackers
- Doing some cool stuff
- Uploading the firmware over the air

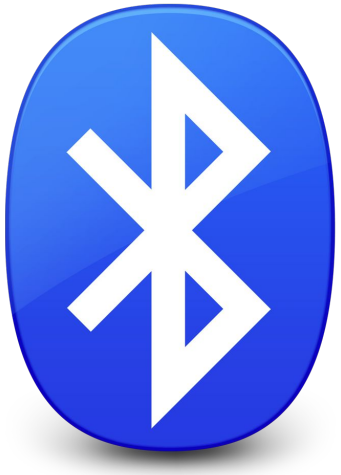
Bluetooth Story...

Bluetooth is a short-range wireless communication protocol and allows devices such as smartphones, headsets, to transfer data and/or voice wirelessly.

Developed in 1994 as a replacement for cables.

Uses 2.4GHz frequency and creates 10 meters radius called

piconet!



And comes Bluetooth Low Energy(4.0)...

Bluetooth low energy aka Bluetooth Smart

- Designed to be power efficient
- Low cost and easy to implement
- Used in sensors, lightbulbs, medical devices, wearables and many other “smart” products.

Bluetooth
SMART

Bluetooth classic vs BLE

Bluetooth Classic	Bluetooth Low Energy

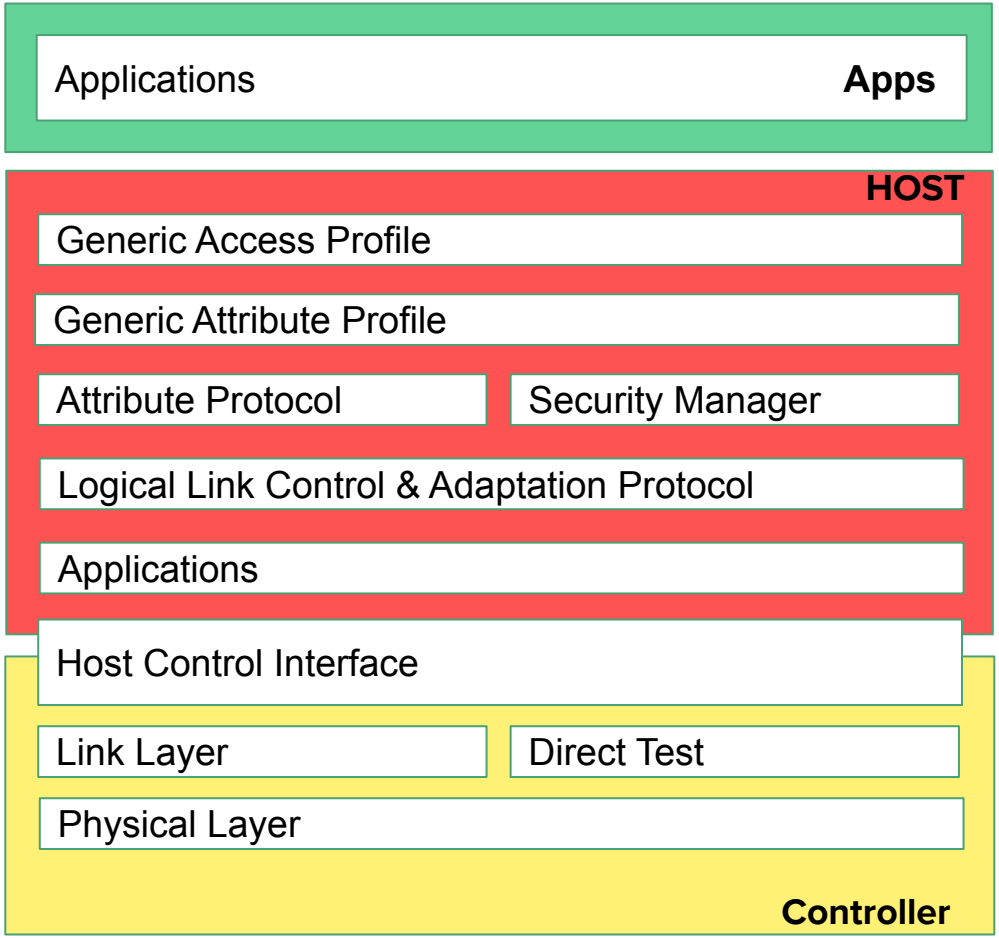
● Great for products that requires continuous streaming of data
● High power consumption
● Faster data rate
● High application throughput
● Best Suited for:
  ○ Headsets, Speakers
  ○ Bluetooth Hotspot etc
	● Great for products that **do not require continuous streaming** of data.
	● Ultra low power consumption
	● Slower Data rate
	● Low application throughput
	● Best Suited for:
	  ○ Home Automation
	  ○ Fitness trackers etc
	
	It is designed to operate in sleep mode and waken up only when connection is initiated. Like maybe your light is on or off or a quick command to turn on or off the light.

Bluetooth Low Energy

Fitness Tracker - BLE Applications

BLE Stack

- Generic Attribute Profile (GATT)
- Generic Access Profile(GAP)



Generic Attribute Profile (GATT)

GATT defines the way that these BLE devices communicate with each (client & server) other using something called **Services** and **Characteristics**.

Here Connections are Exclusive! Means your BLE peripheral can only be connected to one central device at a time! It will stop advertising itself and other devices will no longer be able to see it or connect to it until the existing connection is broken.

Basic Process

1. Select the target
   a. Install Bluez stack, hcitool & gattool
2. Enumerate the **services** and **characteristics**
   a. Do the scan using hcitool
   b. Connect using gatttool
   c. List all the services and characteristics
3. Reverse Engineer the mobile application (if any)
   a. For reverse engineering android application use apktool.
4. Finally do some cool stuff!

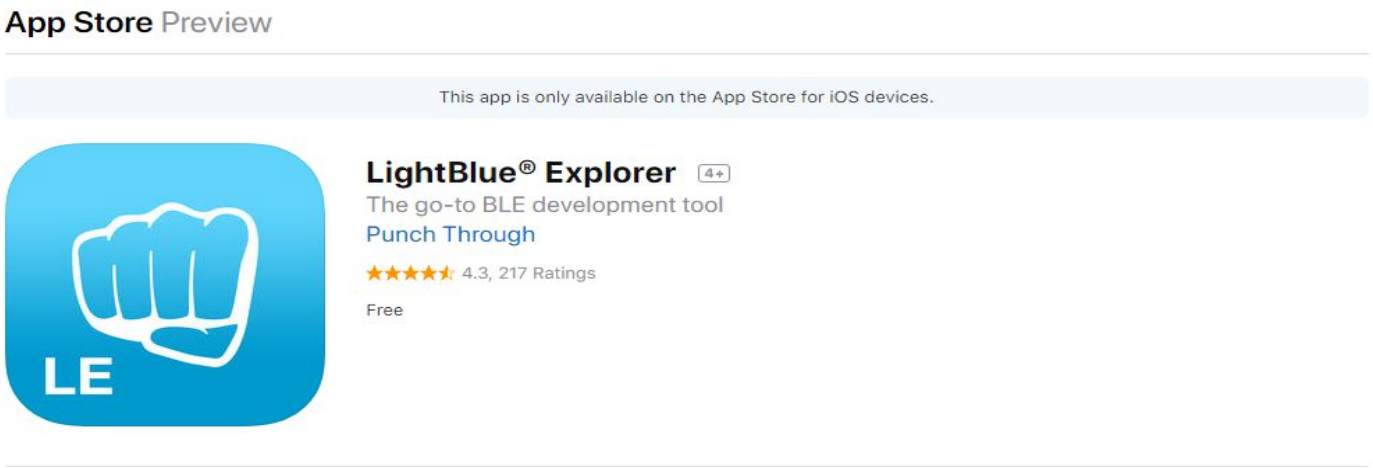
Selecting the target

Goal: Finding the BLE devices near the vicinity

Tools Used: **Bluez, hcitool, gatttool**

Install Bluez: $ sudo apt-get install bluez

Install Hcitool: hcitool comes **preinstalled with bluez stack**



Enumerate the services and characteristics

sudo gatttool -b <BLE ADDRESS> -I

>connect

List down all primary services

> primary

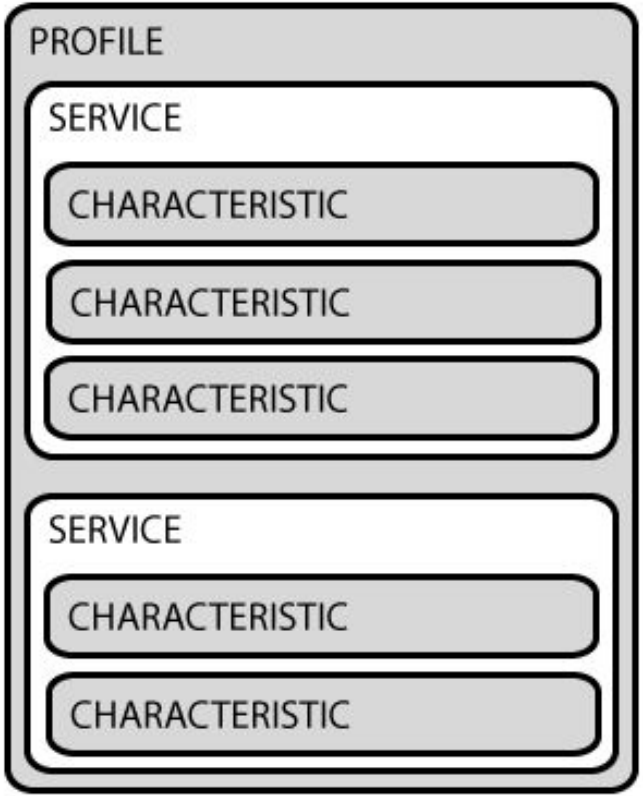
List down all characteristics

> characteristics

Services & Characteristics

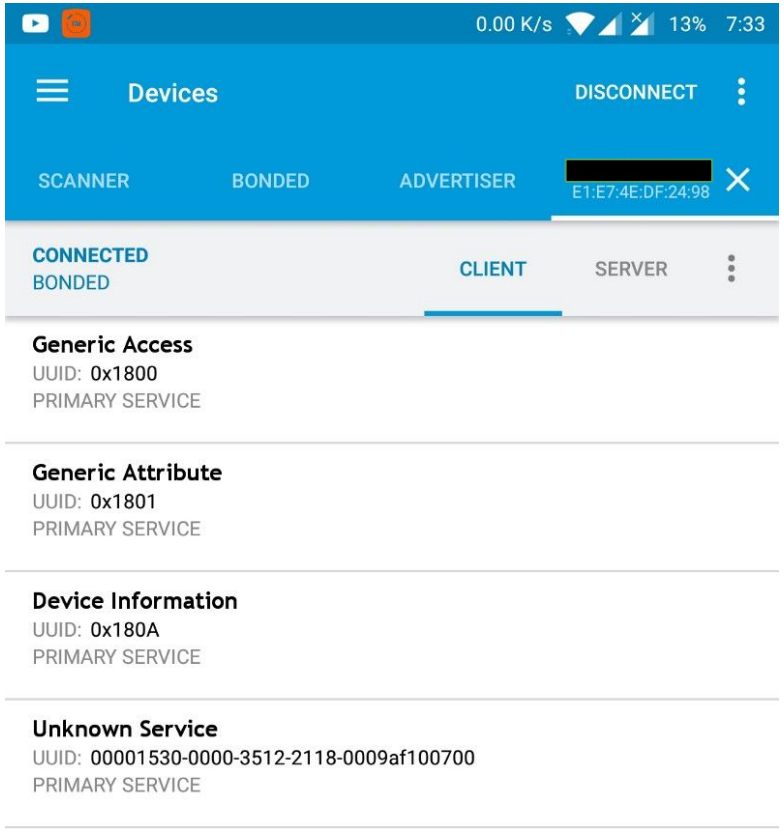
Services: Set of provided features and associated behaviors to interact with the peripheral. Each service contains a collection of characteristics.

Characteristics: Characteristics are defined attribute types that contain a single logical value.



https://learn.adafruit.com/

Services & Characteristics

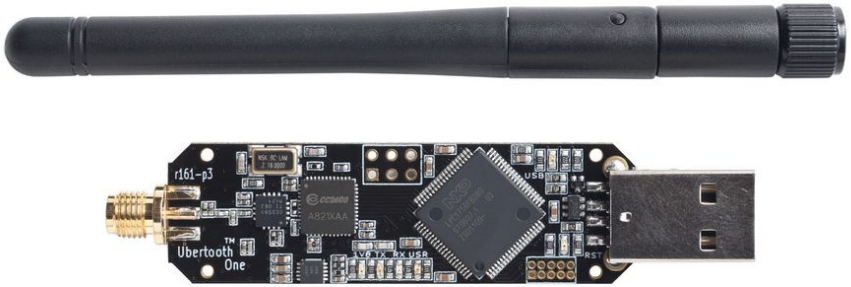


https://learn.adafruit.com/

Sniffing BLE Packets

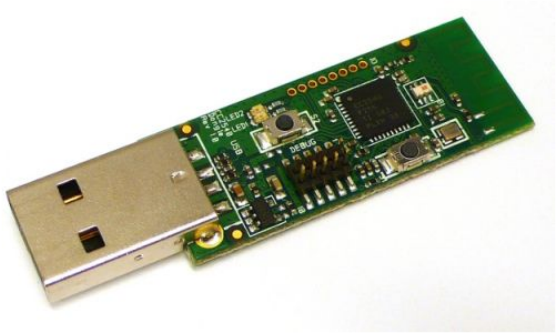
Ubertooth

- Works great for both Classic and BLE
- Open Source Hardware/Software
- About $100



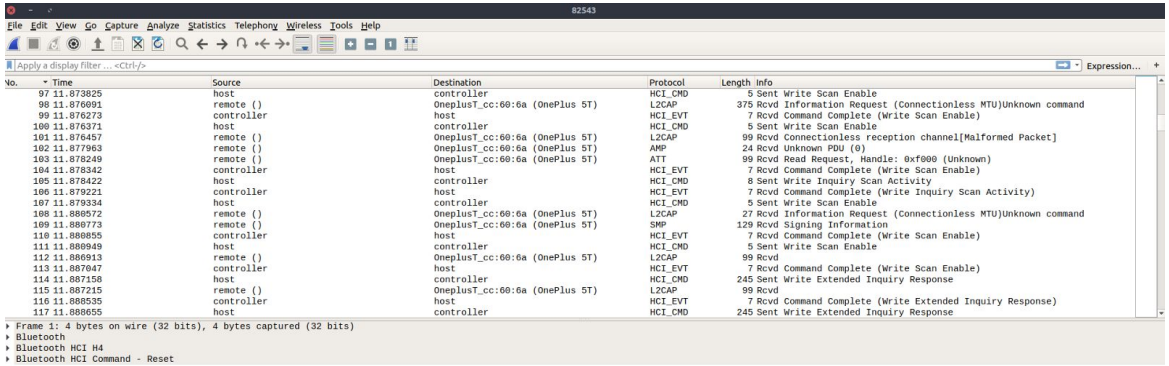
CC2540

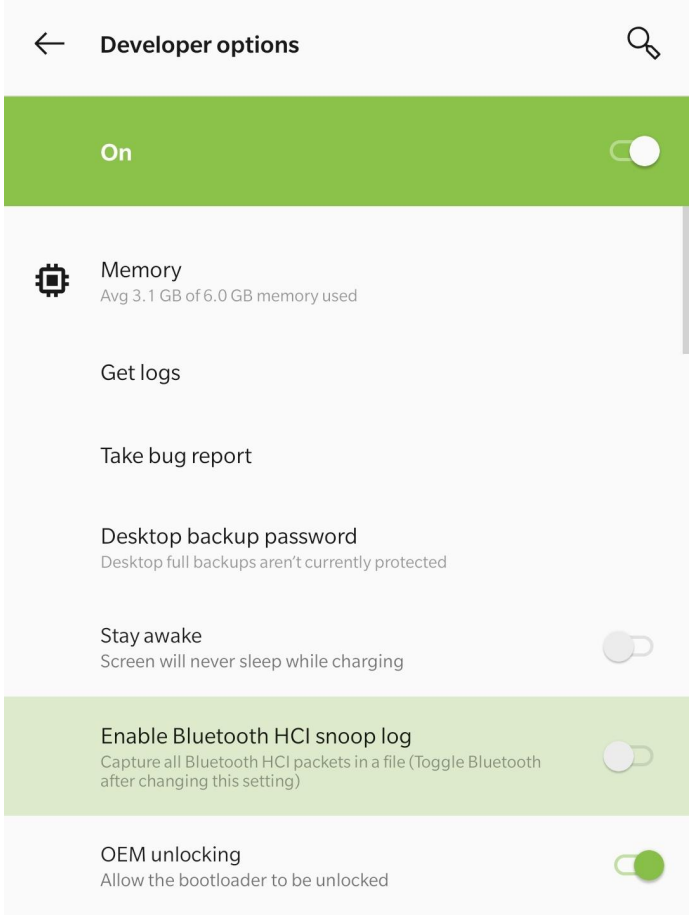
- Cheaper but limited configuration
- About $50



Alternate to Sniffers

- Enable Developer Option
- Enable Bluetooth HCI Snoop Log
- $ adb pull /sdcard/btsnoop_hci.log





Authentication

- Setting on auth notifications (to get a response) by sending 2 bytes request \x01\x00 to the Des.

- Send 16 bytes encryption key to the Char with a command and appending to it 2 bytes \x01\x00 + KEY.

- Requesting random key from the device with a command by sending 2 bytes \x02\x00 to the Char.
- Getting random key from the device response (last 16 bytes).

- Encrypting this random number with our 16 bytes key using the AES/ECB/NoPadding encryption algorithm (from Crypto.Cipher import AES) and send it back to the Char (\x03\x00 + encoded data)

Main Service UUID

0000fee1-0000–1000–8000–00805f9b34fb

Auth Characteristic UUID

00000009–0000–3512–2118–0009af100700

Notification descriptor handle

0x2902

Thanks to Andrey Nikishaev

https://medium.com/@a.nikishaev

Send some Notification? ;)

17:58:37.879	Writing request to characteristic 00002a46-0000-1000-8000-00805f9b34fb
17:58:38.428	Data written to 00002a46-0000-1000-8000-00805f9b34fb, value: (0x) 03-01-48-69
17:58:38.428	"Call, Count: 1, Message: Hi" sent

First Two Byte is Notification Type

01 -> Email

03 -> Call

04 -> Missed Call

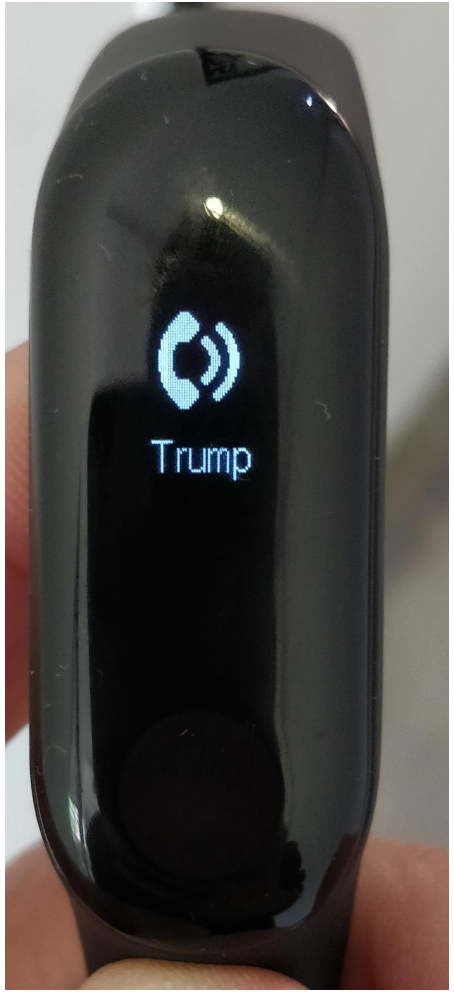
05 -> SMS/MMS

Next Two Byte is numbers of notification

And remaining is the hex value of the notification title that you are sending.

Send some Notification? ;)

```
def send_custom_alert(self, type):
        if type == 5:
            base_value = '\x05\x01'
        elif type == 4:
            base_value = '\x04\x01'
        elif type == 3:
                base_value = '\x03\x01'
        phone = raw_input('Sender Name or Caller ID')
        svc = self.getServiceByUUID('"00001811-0000-1000-8000-00805f9b34fb')
        char = svc.getCharacteristics('00002a46-0000-1000-8000-00805f9b34fb')[0]
        char.write(base_value+phone, withResponse=True)
```


Trump

Firmware

My aim was to display this!

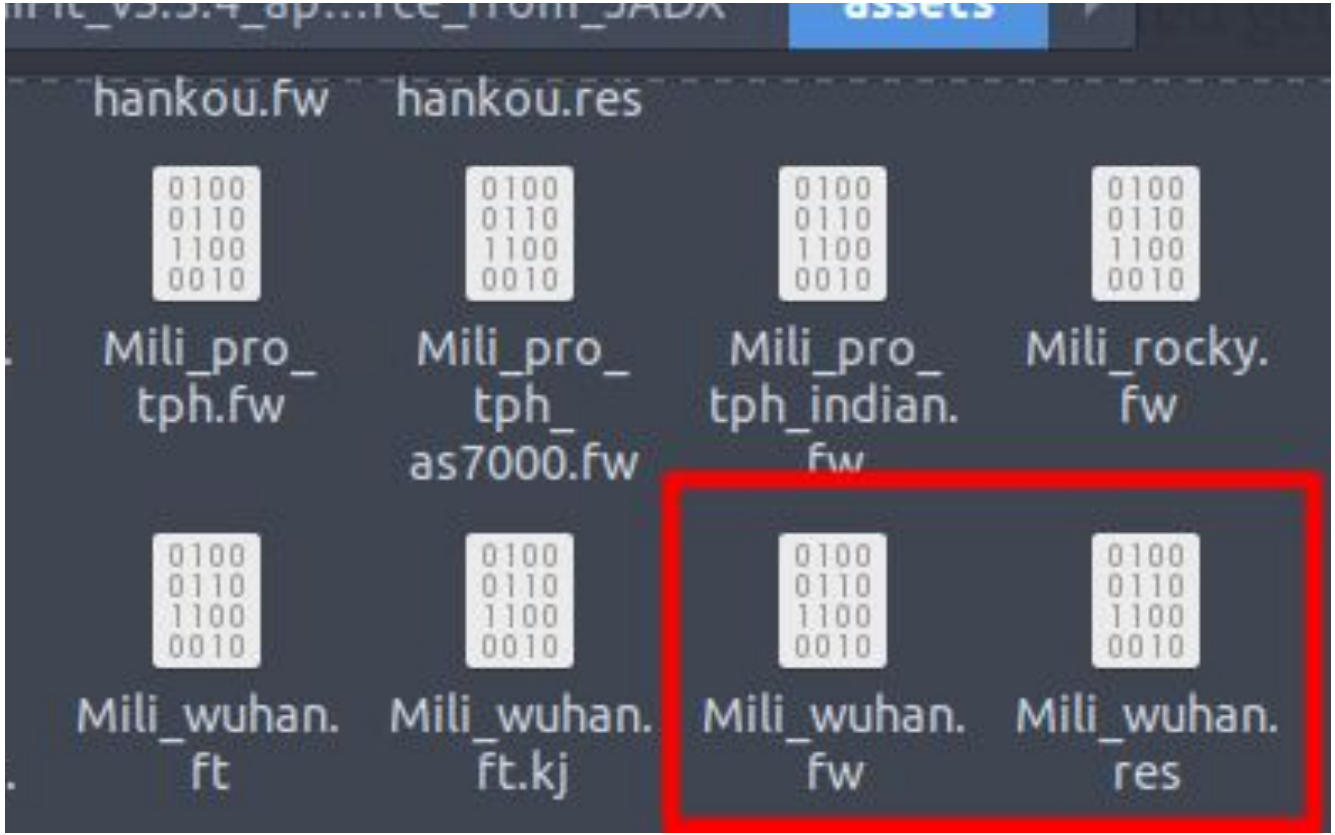
Firmware!!!

A **firmware** is a piece of Software that runs on embedded CPU!

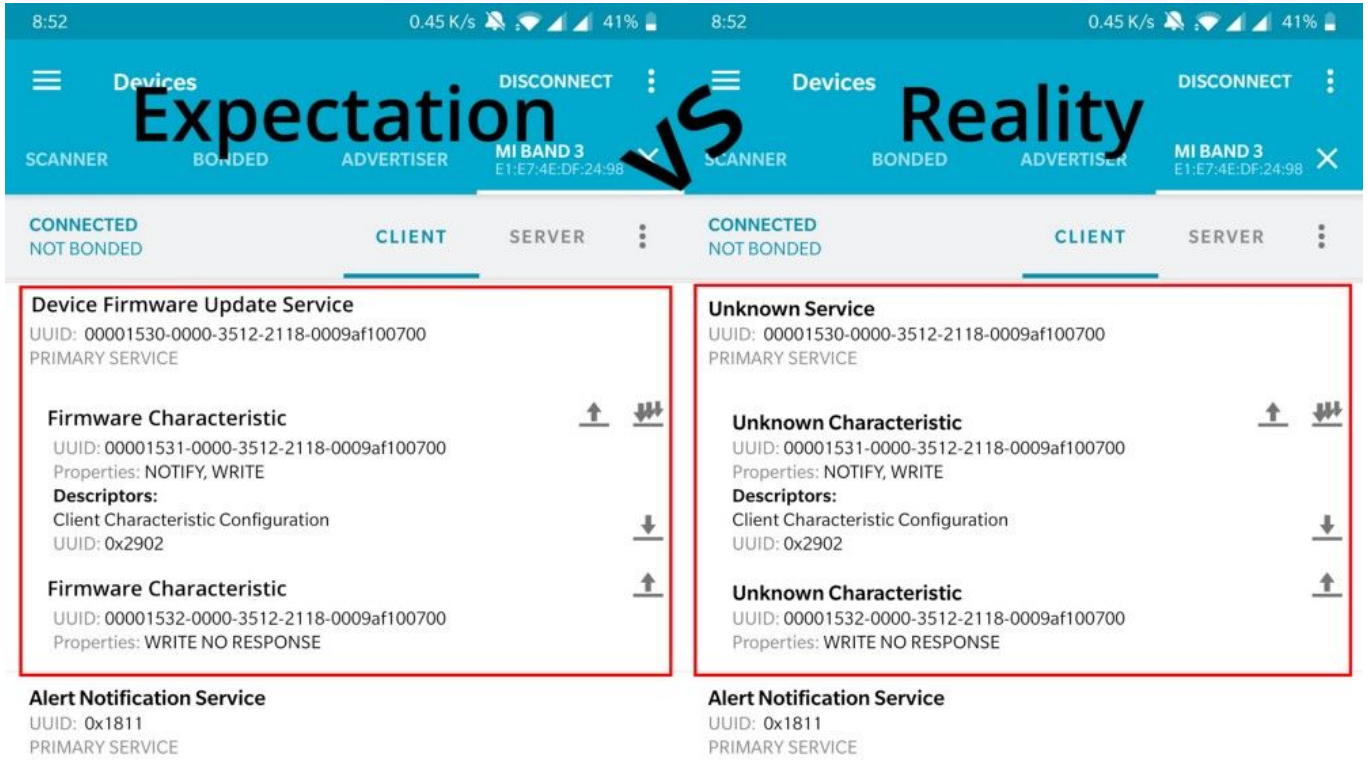
How do I get firmware?
Reverse Engineering the Mobile application maybe? Or during the DFU update?

Let’s reverse engineer the mobile application!

$ apktool d cool_app.apk


hankou.fw
hankou.res
Mili_pro_tph.fw
Mili_pro_tph_as7000.fw
Mili_pro_tph_indian.fw
Mili_rocky.fw
Mili_wuhan.ft
Mili_wuhan.ft.kj
Mili_wuhan.fw
Mili_wuhan.res

Uploading the firmware



How does firmware upload works?

- Initialize the firmware/resource Update On Characteristic 1531 with write command of 4-byte
- **\x01** + fileSize in Hex(3-byte)
- But, for the resource, its **5-byte**!
  \x01 + **fileSize** in Hex(3-byte) + **\x02**
- Last byte **\x02** is for letting the firmware update service know that it's a resource and not the firmware file.

Doesn’t accept 0x5EFAC but accepts 0xAcEF05

How does firmware upload works?

After that,

- Send **\x03** to notify **Start Data,** and you are ready to transfer the Firmware
- It can receive a **maximum of 20 bytes** for a single command. Send **20 bytes** at a time. The firmware/resource has to be written on **Characteristic “1532”**.
- Send **\x00** on characteristic 1531, it’s update sync command

Your firmware is uploaded, but something is missing! **Checksum!!!**

How does firmware upload works?

What is **Checksum?**
Calculated value that is used to determine the integrity of data during the transmission.

BLE does not perform error correction but can only perform error detection.
Bluetooth 5.0 introduces error correction.

How does firmware upload works?

Once the CRC is calculated, write the checksum to Characteristic “1531” of 3 bytes. The checksum must begin with \x04 and your checksum value

\x04 + checksum

If the checksum matches the resource will be accepted and updated. But for firmware, you need to send reboot command as well.

On Characteristic “1531” send \x05 for the reboot.

And yes, the firmware update is done!

Device Name
UUID: 0x2A00
Properties: READ
Value: Jasper X

Appearance
UUID: 0x2A01
Properties: READ

Peripheral Preferred Connection Parameters
UUID: 0x2A04
Properties: READ

Generic Attribute
UUID: 0x1801
PRIMARY SERVICE

Device Information
UUID: 0x180A
PRIMARY SERVICE

Serial Number String
UUID: 0x2A25
Properties: READ

Hardware Revision String
UUID: 0x2A27
Properties: READ

Software Revision String
UUID: 0x2A28
Properties: READ
Value: V9.9.9.9

And what about the skull Icon? ;)

Q&A

More about this hack is on Medium & Github!

https://medium.com/@yogeshojha

https://github.com/yogeshojha/MiBand3/

```
MiBand MAC: E1:E7:4E:DF:24:98

Select an option
  1 - View Band Detail info
  2 - Send a High Prority Call Notification
  3 - Send a Medium Prority Message Notification
  4 - Send a Message Notification
  5 - Send a Call Notification
  6 - Change Date and Time
  7 - Send a Missed Call Notification
  8 - Get Heart BPM
  9 - DFU Update
  10 - Exit
```