OPEN SOURCE SUMMIT
China 2019
Unfit Story of Fitness Trackers: Hacking the BLE devices

Yogesh Ojha

- Cyber Security Analyst @ TCS, India
- IOT & Mobile Application Security
You can expect to learn about:

- Basic Understanding of Bluetooth
- Bluetooth Classic vs Bluetooth Low Energy
- BLE Stack
- Capturing BLE Packets/BLE MiTM
- Reverse Engineering the Mobile Applications of Fitness trackers
- Uploading the firmware over the air
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!
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 classic vs BLE

Bluetooth Classic
- Great for products that require continuous streaming of data
- High power consumption
- Faster data rate
- High application throughput
- Best Suited for:
  - Headsets, Speakers
  - Bluetooth Hotspot etc

Bluetooth Low Energy
- 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 (4.0)
Fitness Tracker - BLE Applications
BLE Stack

- Generic Attribute Profile (GATT)
- Generic Access Profile (GAP)
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.
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.
Services & Characteristics

Generic Access
UUID: 0x1800
PRIMARY SERVICE

Generic Attribute
UUID: 0x1801
PRIMARY SERVICE

Device Information
UUID: 0x180A
PRIMARY SERVICE

Unknown Service
UUID: 00001530-0000-1575-0000-000000000000
PRIMARY SERVICE
Basic Process

0. Select the target
   a. Install Bluez stack, hcitool & gattool

1. Enumerate the **services** and **characteristics**
   a. Do the scan using hcitool
   b. Connect using gatttool
   c. List all the services and characteristics

2. Reverse Engineer the mobile application (if any)
   a. For reverse engineering android application use apktool.

3. Finally do some cool stuff!
0. 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
Scanning for BLE Devices

yogesh@yogesh:~$ sudo hcitool lescan
LE Scan ...
E1:E7:4E:DF:24:98 (unknown)
E1:E7:4E:DF:24:98 Mi Band 3
1. Enumerate the services and characteristics

```
sudo gatttool -b <BLE ADDRESS> -I

   >connect

List down all primary services

   > primary

List down all characteristics

   > characteristics
```
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`
3 devices, out of 5 devices that I tested, did not implement link layer encryption.

2 devices, out of 5 devices that I tested, did not have authentication!!!
Send some Notification? ;)

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.
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)
My aim was to display this!
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
Uploading the firmware
How does firmware upload works? For this fitness tracker

- 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? For this fitness tracker

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? For this fitness tracker

Once the CRC is calculated, write the checksum to Characteristic “XXXX” 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 “XXXX” send \x05 for the reboot.

And yes, the firmware update is done!
Device Name
UUID: 0xA00
Properties: READ
Value: Jasper X

Appearance
UUID: 0xA01
Properties: READ

Peripheral Preferred Connection Parameters
UUID: 0xA04
Properties: READ

Generic Attribute
UUID: 0x1801
PRIMARY SERVICE

Device Information
UUID: 0x180A
PRIMARY SERVICE

Serial Number String
UUID: 0xA25
Properties: READ

Hardware Revision String
UUID: 0xA27
Properties: READ

Software Revision String
UUID: 0xA28
Properties: READ
Value: V9.9.9.9
What about the skull icon? ;)
More about this hack is on Medium & Github!
https://medium.com/@yogeshojha
https://github.com/yogeshojha/MiBand3/