

# 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 classic vs BLE

Bluetooth Classic	Bluetooth Low Energy
<ul style="list-style-type: none"><li>• Great for products that requires continuous streaming of data</li><li>• High power consumption</li><li>• Faster data rate</li><li>• High application throughput</li><li>• Best Suited for:<ul style="list-style-type: none"><li>◦ Headsets, Speakers</li><li>◦ Bluetooth Hotspot etc</li></ul></li></ul>	<ul style="list-style-type: none"><li>• Great for products that <b>do not require continuous streaming</b> of data.</li><li>• Ultra low power consumption</li><li>• Slower Data rate</li><li>• Low application throughput</li><li>• Best Suited for:<ul style="list-style-type: none"><li>◦ Home Automation</li><li>◦ Fitness trackers etc</li></ul></li></ul> <p data-bbox="981 754 1837 926">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.</p>

# Bluetooth Low Energy



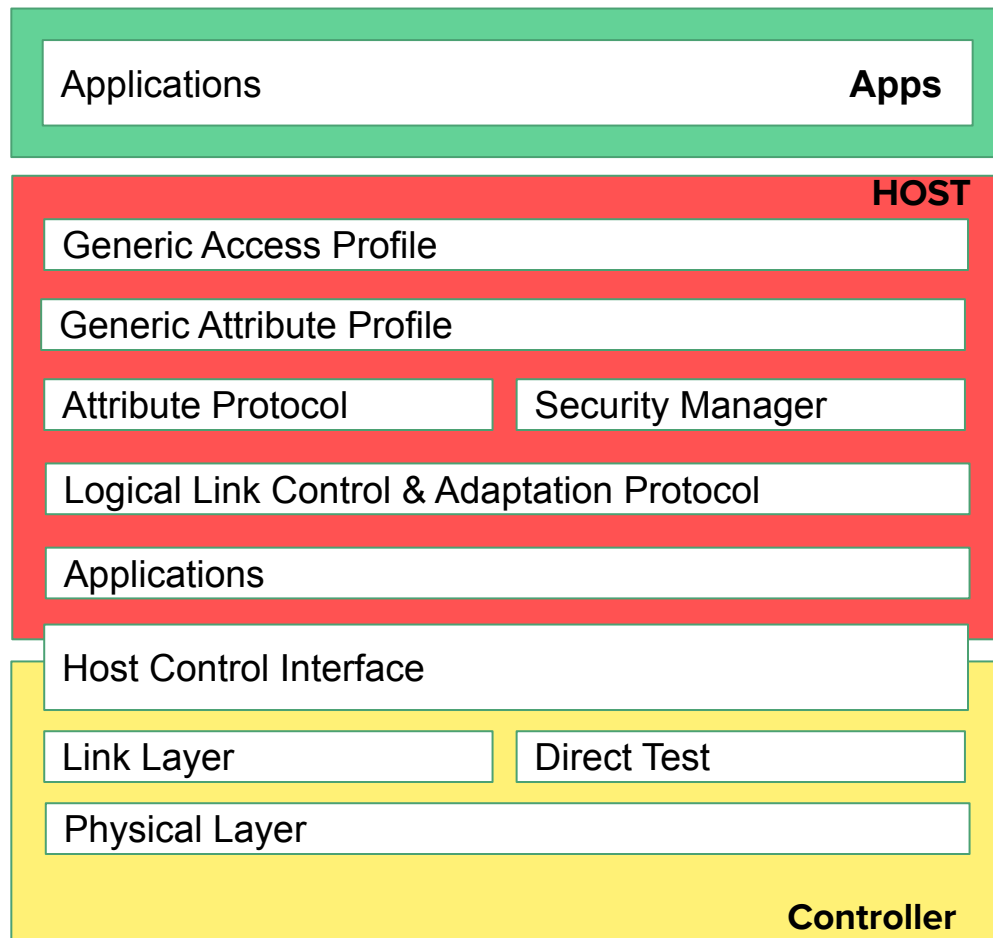
# Fitness Tracker - BLE Applications





# 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 & gatttool
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**

## App Store Preview

This app is only available on the App Store for iOS devices.



### LightBlue® Explorer 4+

The go-to BLE development tool

Punch Through

★★★★★ 4.3, 217 Ratings

Free

# Enumerate the services and characteristics

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

```
>connect
```

## **List down all primary services**

```
> primary
```

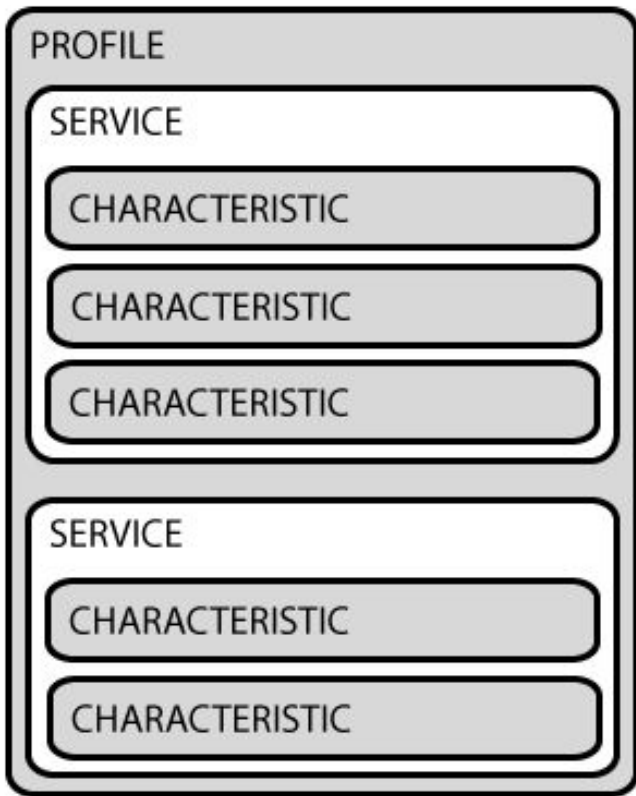
## **List down all characteristics**

```
> characteristics
```

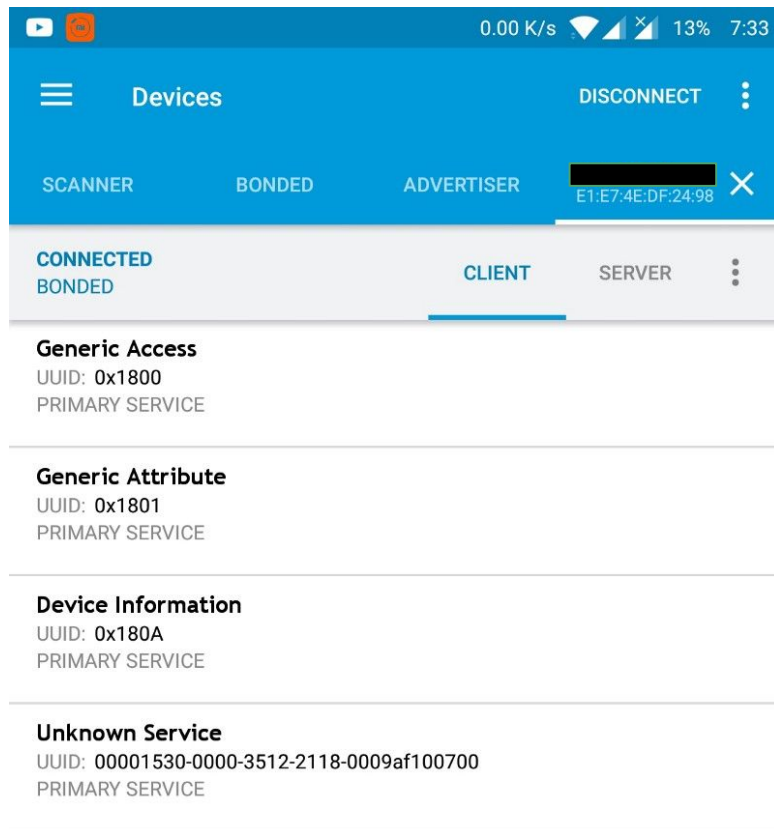
# Services & Characteristics

**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.



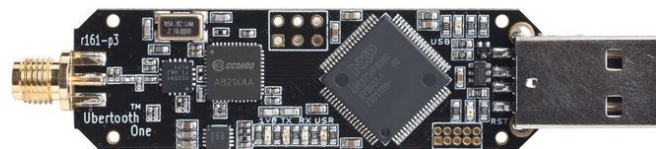
# Services & Characteristics



# Sniffing BLE Packets

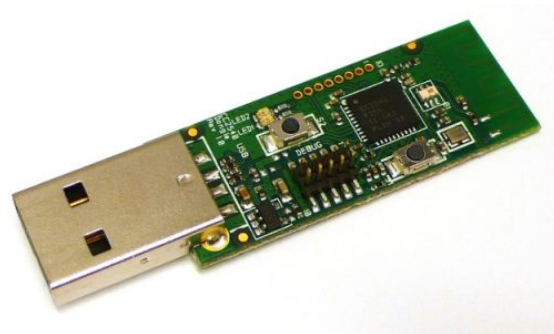
## Ubertooth

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



## CC2540

- Cheaper but limited configuration
- About \$50





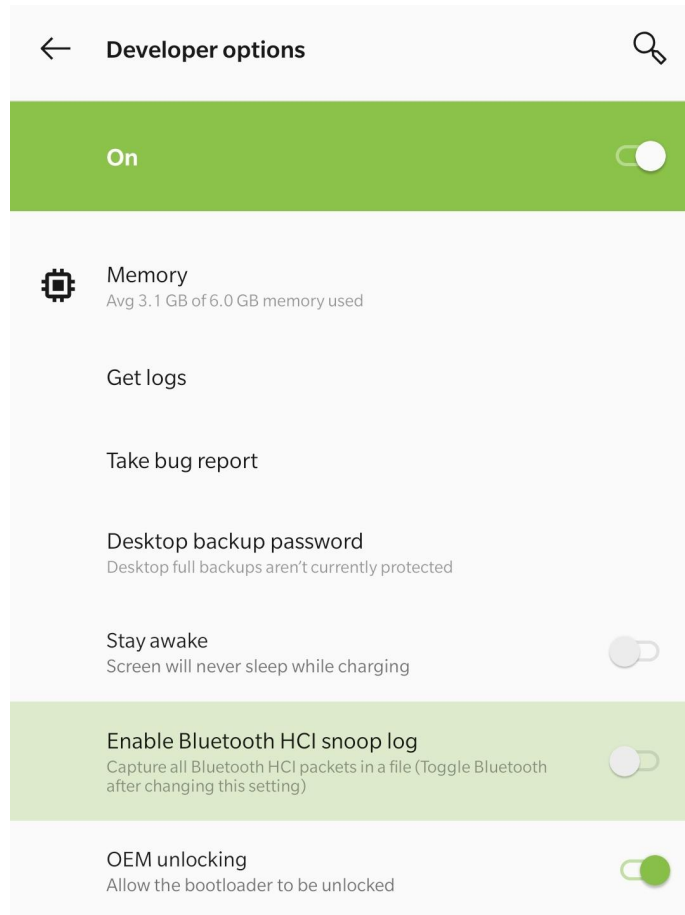
# Alternate to Sniffers

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

Wireshark packet capture analysis of Bluetooth HCI commands. The packet list shows 17 packets, with the last one being a 'Frame 1: 4 bytes on wire (32 bits), 4 bytes captured (32 bits) on Bluetooth HCI H4'. The packet details pane shows the 'Bluetooth HCI Command - Reset' structure.

No.	Time	Source	Destination	Protocol	Length	Info
97	11.873825	host	controller	HCI_CMD	9	5 Sent Write Scan Enable
98	11.876691	remote ()	OnePlusT_cc:00:60:6a (OnePlus 5T)	L2CAP	375	Read Information Request (Connectionless MTU)Unknown command
99	11.876273	controller	host	HCI_EVT	7	Read Command Complete (Write Scan Enable)
100	11.876371	host	controller	HCI_CMD	9	5 Sent Write Scan Enable
101	11.876457	remote ()	OnePlusT_cc:00:60:6a (OnePlus 5T)	L2CAP	99	Read Connectionless reception channel[Malformed Packet]
102	11.877963	remote ()	OnePlusT_cc:00:60:6a (OnePlus 5T)	AMP	24	Read Unknown PDU (0)
103	11.878249	remote ()	OnePlusT_cc:00:60:6a (OnePlus 5T)	ATT	99	Read Read Request, Handle: 0x0000 (Unknown)
104	11.878342	controller	host	HCI_EVT	7	Read Command Complete (Write Scan Enable)
105	11.878422	host	controller	HCI_CMD	8	5 Sent Write Inquiry Scan Activity
106	11.879221	remote ()	OnePlusT_cc:00:60:6a (OnePlus 5T)	L2CAP	7	Read Command Complete (Write Inquiry Scan Activity)
107	11.879334	host	controller	HCI_CMD	5	5 Sent Write Scan Enable
108	11.880572	remote ()	OnePlusT_cc:00:60:6a (OnePlus 5T)	L2CAP	27	Read Information Request (Connectionless MTU)Unknown command
109	11.880773	remote ()	OnePlusT_cc:00:60:6a (OnePlus 5T)	SNP	125	Read Signaling Information
110	11.880855	controller	host	HCI_EVT	7	Read Command Complete (Write Scan Enable)
111	11.880949	host	controller	HCI_CMD	5	5 Sent Write Scan Enable
112	11.880913	remote ()	OnePlusT_cc:00:60:6a (OnePlus 5T)	L2CAP	99	Read Read Request, Handle: 0x0000 (Unknown)
113	11.887047	controller	host	HCI_EVT	7	Read Command Complete (Write Scan Enable)
114	11.887158	host	controller	HCI_CMD	245	5 Sent Write Extended Inquiry Response
115	11.887215	remote ()	OnePlusT_cc:00:60:6a (OnePlus 5T)	L2CAP	99	Read Read Request, Handle: 0x0000 (Unknown)
116	11.888535	controller	host	HCI_EVT	7	Read Command Complete (Write Extended Inquiry Response)
117	11.888655	host	controller	HCI_CMD	245	5 Sent Write Extended Inquiry Response

Frame 1: 4 bytes on wire (32 bits), 4 bytes captured (32 bits) on Bluetooth  
 Bluetooth  
 Bluetooth HCI H4  
 Bluetooth HCI Command - Reset



# 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
```

Ale  
UU  
PR


## First Two Byte is Notification Type

- 01 -> Email
- 03 -> Call
- 04 -> Missed Call
- 05 -> SMS/MMS

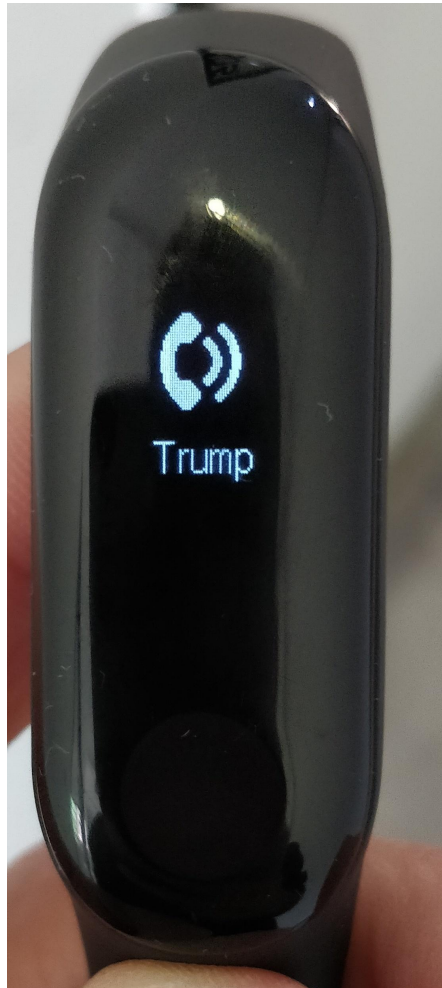
## Next Two Byte is numbers of notification

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)
```



# Firmware

My aim was to display this!



# Firmware!!!

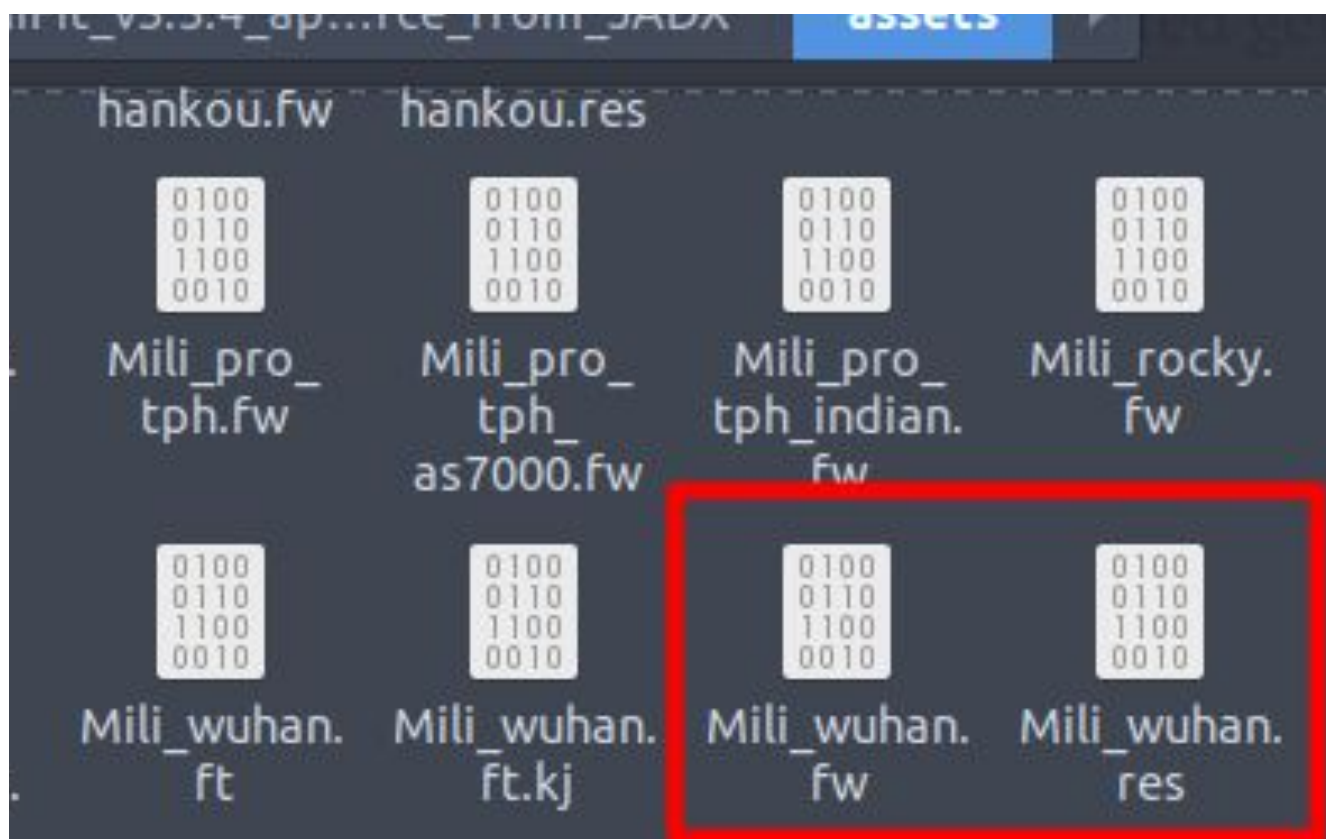
A **firmware** is a piece of Software that runs on embedded CPU!

## 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
```





# Uploading the firmware

Expectation vs Reality

The screenshot displays a mobile application interface for managing a device, specifically 'MI BAND 3' with MAC address 'E1:E7:4E:DF:24:98'. The interface is split into two panels: 'Expectation' on the left and 'Reality' on the right, separated by a large 'VS' graphic. Both panels show the device is 'CONNECTED' and the 'CLIENT' tab is selected. The status bar at the top shows the time as 8:52, a speed of 0.45 K/s, and 41% battery.

**Expectation Panel:**

- Device Firmware Update Service**  
UUID: 00001530-0000-3512-2118-0009af100700  
PRIMARY SERVICE
- Firmware Characteristic**  
UUID: 00001531-0000-3512-2118-0009af100700  
Properties: NOTIFY, WRITE
- Descriptors:**  
Client Characteristic Configuration  
UUID: 0x2902
- Firmware Characteristic**  
UUID: 00001532-0000-3512-2118-0009af100700  
Properties: WRITE NO RESPONSE
- Alert Notification Service**  
UUID: 0x1811  
PRIMARY SERVICE

**Reality Panel:**

- Unknown Service**  
UUID: 00001530-0000-3512-2118-0009af100700  
PRIMARY SERVICE
- Unknown Characteristic**  
UUID: 00001531-0000-3512-2118-0009af100700  
Properties: NOTIFY, WRITE
- Descriptors:**  
Client Characteristic Configuration  
UUID: 0x2902
- Unknown Characteristic**  
UUID: 00001532-0000-3512-2118-0009af100700  
Properties: WRITE NO RESPONSE
- Alert Notification Service**  
UUID: 0x1811  
PRIMARY SERVICE

# 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 Priority Call Notification
- 3 - Send a Medium Priority 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