Stay fit: Hack a Jump Rope

Axelle Apvrille

2023-02-23

Stay fit: Hack a Jump Rope

Ph0wn is a Capture The Flag (CTF) event on the French riviera. It focuses on hacking challenges for IoT, and therefore, we are always on the look out for intriguing, interesting (but not too expensive) connected objects to hack. During the last edition, we decided to hack a connected jump rope.

The jump rope is meant to help you remain fit. It communicates via Bluetooth Low Energy (BLE) to a smartphone and counts the number of jumps, time of effort, speed etc.

In this presentation, we are going to see:

1. How to reverse engineer such a device. Initially, we don't know how the connected jump rope works, and there is no developer documentation. We need to understand how to send commands to the rope, how to get results etc.
2. How to create a CTF challenge based on the device. Indeed, CTF challenges have special requirements such as protecting the flag, preventing a team B from re-using the work team A did etc.

While some parts are specific to this connected jump rope, many concepts presented in this talk are generic to hacking any Bluetooth Low Energy device.

NB. We will not cover how to *solve* the challenge as write-ups for this challenge exist here.

Presentation of the jump rope

The jump rope is manufacturer by *Renpho* under the simple product name of "Smart Jump Rope". It costs approximately 30 euros. This presentation covers model *R-Q001*. The user manual can be downloaded from [RENPHO]. The rope communicates with a smartphone application named *Renpho Fit* via BLE. Product description advertises use of 3 *Hall sensors* to detect accurately the number of jumps. For more information on Hall sensors, we redirect the reader to Wikipedia [HALL] (basically, it detects presence and magnitude of a magnetic field).

The jump rope has a small LCD screen which displays the current status (e.g. number of jumps, or time etc). The jump rope basically offers 3 different operational modes:

1. **Free Jump** Mode. You just start jumping and the LCD displays the current number of jumps.
2. **Time Countdown** Mode. Your goal is to jump for x minutes. The LCD displays the remaining time.
3. **Numbers Countdown** Mode. Your goal is to jump x times. The LCD displays the remaining number of jumps to do.

Whatever mode, the app is able to display all sorts of data like number of jumps, duration, average or peak speed, stumbles, keep history etc. By default, the app has nearly no interaction with Internet: logging in one's fitness account is not mandatory, it's possible to use the app in a "guest" mode. While the app does offer some basic optional community interaction (e.g ranking among the community), this part - i.e communication between the app and remote servers on Internet - has not been studied. This presentation focuses on communication between the jump rope device and the app.

The jump rope is powered by batteries, it works fine and this presentation bears no criticism on the device.

Figure 1: Smart rope sensors - this image is part of the public description of the jump rope on renpho.com

Reversing a BLE device

There are basically 2 strategies to reverse any BLE device:

1. **Network**. While tools like BLE Sniffer, Ubertooth allow you to sniff BLE traffic, they are not always easy to set up. If we don't want to use a specific sniffing device, another option is to use existing BLE **notifications** (and *indications*).
2. **Application**. The BLE device typically communicates with a mobile application. The reverse engineering of the mobile app reveals lots (if not all) information to discuss with the device.

BLE Notifications

A BLE device is organized in *services* (either a standard service such as "Generic Access" or a device specific device), and services group different *characteristics*. Various operations are possible on characteristics: read, write, notify, indicate. For example, the "Generic Access" service has a characteric named "Device Name" which supports the "read" operation. In practice, this means we can query the device to read its BLE "Device Name".

Some characteristics have values which are meant to evolve: a counter for example. Developers will typically support the "notify" operation on those. To receive notifications, one must first *enable* them, and then whenever the value changes, the BLE device sends a *notification*.

This is typically interesting for the connected jump rope. Obviously, at some point, the jump rope must let the application know the jump counter has increased.

We get connect to the jump rope (e.g with `bluetoothctl`). The tool enumerates existing services and characteristics. We can search for existing characteristics which support the *notify* operation (see `notify` flag)

```
[RENPHO-ROPE-R1]# attribute-info 00005303-0000-0041-4c50-574953450000
Characteristic - Vendor specific
    UUID: 00005303-0000-0041-4c50-574953450000
    Service: /org/bluez/hci0/dev_DF_E5_34_0E_42_7D/service000e
    Notifying: no
    Flags: read
    Flags: notify
```

To enable notifications, we have to modify its "Client Characteristic Configuration Descriptor" (CCCD). The way

to do this depends on the tool: with the old `gatttool` you need to write "0100" to the corresponding descriptor handle, with `bluetoothctl`, you select the attribute (`select-attribute`) and then issue `acquire-notify`.

```
# select-attribute /org/bluez/hci0/dev_DF_E5_34_0E_42_7D/service000e/char0011
[RENPHO-ROPE-R1:/service000e/char0011]# acquire-notify
[CHG] Attribute /org/bluez/hci0/dev_DF_E5_34_0E_42_7D/service000e/char0011 NotifyAcquired: yes
AcquireNotify success: fd 7 MTU 131
```

Once this is done, the connected jump rope sends you a Bluetooth packet each time something changes on those characteristics. For example, this is sent after jumping 3 times:

```
Notification handle = 0x0012 value: ad 2b 00 40 00 11 00 00 00 00 00 00 00 00 01 00 00 00 00 00
Notification handle = 0x0012 value: af 2b 01 01 5e 46 2f 9a
Notification handle = 0x0012 value: ad 2c 00 40 00 11 00 00 00 00 01 00 00 00 02 00 00 00 00 00
Notification handle = 0x0012 value: af 2c 01 01 5e 46 5e 8c
Notification handle = 0x0012 value: ad 2d 00 40 00 11 00 00 00 00 02 00 00 00 03 00 00 00 00 00
Notification handle = 0x0012 value: af 2d 01 01 5e 46 0d 84
```

After a few tries (e.g jump once or more, try time lapse etc), we understand packets are sent in a group. The first packet of a group always begins with `ad`. Middle packets (if any) begin with `ae`, and final packets begin with `af`. The second byte of each packet is a group counter (e.g `2b`). All packets of the same group share the same group counter. The third byte is a counter within the group: the first packet starts at 0.

Then, the payload of the packet depends of each mode/action. For a free jump, based on the packets above (and a few more) we understand the format is `40 00 11 xx xx xx xx TL JJ JJ JJ JJ` where:

- `40` identifies the packet type
- `TL` contains the time length
- `JJ JJ JJ JJ` contains the jump counter

With patience and methodology, the analysis of transmitted packets in each situation reveals lots information of packet format. Yet, it may be long to reverse all communication this way as you have to reproduce each case. This is where *reversing the application* comes handy.

Reversing the application

I have personally more experience in reversing Android applications, and consequently find this solution easier and fruitful.

Logcat A low hanging fruit comes from Android system logs. The application openly logs lots of information in logcat and they can be easily read via `logcat` with the log tag `TAG`:

```
0-28 16:57:25.322 27999  6570 E AndroidBLE: [10854] TAG: onNotifySuccess: RENPHO-ROPE-R1
10-28 16:57:25.324 27999  6570 E TAG     : HEX=0100061939101C0A1505FD>>>>>>>>>>05FD
10-28 16:57:25.353 27999  6570 I TAG     : onChanged==data:[81, 00, 03, 00, 28, 00, 4e, 00]
10-28 16:57:25.460 27999  6570 I TAG     : onChanged==data:[83, 00, 05, 00, 7d, 2b, c0, ab, f0, 71]
10-28 16:57:25.551 27999  6570 I TAG     : onChanged==data:[84, 00, 02, 00, 00, de, 75]
```

Among other data, the application logs the full BLE packets (see line `HEX=`) - exactly what we captured through network analysis. An emphasis is put on the last 2 bytes (see above `>>>>>>>>>>05FD`): we'll see later those are CRC bytes.

Reversing commands to the rope The application is fortunately not obfuscated and implemented by a skilled and clean Android developer. Many comments (in Chinese) are left in the code and help its understanding. Class `com.renpho.fit.App` extends the Android `Application` class and is the main entry point for the code.

```
public void onCreate() {
        super.onCreate();
        this.appViewModelStore = new ViewModelStore();
        Application application0 = (Application)this;
        App.appViewModel = (AppViewModel)new ViewModelProvider(((ViewModelStoreOwner)this), ((Factory)Andr
        App.app = this;
        MultiLanguages.init(application0);
```

```
        RetrofitManage.setUrl("https://admin.renpho.com/");
        RetrofitManage.setmApiProvider(((apiProvider)App.onCreate.1.INSTANCE));
        Context context0 = (Context)this;
        MMKV.initialize(context0);
        EventBus.getDefault().register(this);
        this.initBle();
```

This piece of code tells us the application communicates with a remote web server https://admin.renpho.com using the common Retrofit SDK, which relies on OkHttp for HTTP communication. As we want to focus on Bluetooth communication, we'll set this aside.

The Bluetooth initialization method contains the UUIDs for the jump ropes services:

1. `00005301-0000-0041-4C50-574953450000` is mentioned in a variable of the `App` class as `RENPHOFIT_SERVICE`, so we don't even have to guess its use.
2. `0000fe59-0000-1000-8000-00805f9b34`, same, is mentioned as the OTA service (over-the-air, for updates).

```
private final void initBle() {
        ScanFilter scanFilter0 = new ScanFilter.Builder().setServiceUuid(ParcelUuid.fromString("00005301-0
        ScanFilter scanFilter1 = new ScanFilter.Builder().setServiceUuid(ParcelUuid.fromString("0000fe59-0
        ArrayList arrayList0 = new ArrayList();
        arrayList0.add(scanFilter0);
        arrayList0.add(scanFilter1);
        Ble.options().setLogBleEnable(true).setThrowBleException(true).setLogTAG("AndroidBLE").setAutoConn
    }
```

Notice the call to `setLogBleEnable(true)` which enables BLE logs, under tag name "AndroidBLE" (`setLogTAG`). We have already seen such logs in logcat.

The code also specifies one characteristic supporting write operations (`setUuidWriteCha`), also referred in the code as `RENPHOFIT_PROPERTIES_WRITE`, and one for notifications (`setUuidNotifyChat`) referred to as `RENPHOFIT_PROPERTIES_NOTIFY`.

This layout of services / characteristics can be easily confirmed with `bluetoothctl` or a BLE application like nRF Connect.

In the decompiled code, we quite naturally head to the `com.renpho.fit.ble` namespace. It contains an absolutely *central* class named `Command` which is going to answer most of our reverse engineering questions. For example, the `countDownTimeTargetValue()` method below details the structure of a packet to initiate the *Time Countdown Mode*:

```
public static byte[] countDownTimeTargetValue(int v) {
        byte b = DeviceUtil.inToByteFour(v);
        byte b1 = DeviceUtil.inToByteThree(v);
        byte b2 = DeviceUtil.inToByteTwo(v);
        byte b3 = DeviceUtil.inToByteOne(v);
        byte[] arr_b = {2, 0, 5, (byte)0x82, b, b1, b2, b3};
        byte[] arr_b1 = CRCCheckUtil.hexStringInputToBytes(DeviceUtil.Bytes2HexString(arr_b));
        byte[] arr_b2 = {2, 0, 5, (byte)0x82, b, b1, b2, b3, arr_b1[0], arr_b1[1]};
        Log.e("TAG", "HEX=" + DeviceUtil.Bytes2HexString(arr_b2) + ">>>>>>>>>>" + CRCCheckUtil.getOuputHex
        return arr_b2;
    }
```

The command will begin by `02 00 05 82`, then it will be followed by the target number of seconds (`v`) on 4 bytes (the first 4 lines respectively grab each byte of the integer). Finally, the remaining 2 bytes are a CRC. The CRC16 it uses is known as CRC-16/MODBUS.

With this information, we are able to initiate a jump session of 3 minutes with `bluetoothctl`: (1) we select the write characteristic, (2) we send the packet after computing the correct CRC.

```
# select-attribute 00005302-0000-0041-4c50-574953450000
# write "0x02 0x00 0x05 0x82 0x00 0x00 0x00 0xB4 0xEE 0xB9"
```

See the Appendix for more commands we are able to reverse the same way: trigger the Free Jump mode, set buzzer on/off, stop a current session etc.

During Ph0wn CTF, we were asking participants to initiate a Numbers Count Down mode of 1337 jumps. This is corresponds to packet `0200058100000539db3e`: `0x81` indicates the Numbers Count Down mode, `00 00 05 39` is 0x539=1337 jumps, and `db 3e` is the computed CRC.

Reversing answers of the rope Although the challenge is solved (and even further as we know how to initiate several other commands), we can reverse the application even further to understand the format of answers from the jump rope.

Class `com.renpho.fit.ble.BleDataHandle` handles incoming packets from the rope. We confirm packets are grouped in what the code calls a *memory group* (`memGroup`, `mGroup`). The packet header is `AD` for the first packets, then `AE` (optional), and finally `AF` for the last packet.

```
private static int bodyNum = 0xAE;
private ArrayList dataArray;
private static int endNum = 0xAF;
private int mGroup;
...
private static int startNum = 0xAD;
```

We remind that, as we saw during network analysis, packet header is followed by a Group counter byte, and then a counter within the group. The payload is sent over 2 or more packets, so basically we have something like `AD 04 00 PP PP .. + AE 04 01 PP PP .. + AF 04 02 PP PP .. CC CC` for a memory group 4 containing 3 packets, with payload `PP PP PP ....` Note the payload ends with CRC16/MODBUS `CC CC`.

A close look at the code details the expected payload. For example, the piece of code below shows 2 important types of responses: RopeData packets, which begin with byte `A0` and FitData packets which starts with `40 00 11`:

```
                case 0xA0: {
                    RopeData ropeData0 = RopeData.getRopeData(arr_b2);
                    EventBus.getDefault().post(ropeData0);
                }
            }
        }
        else if(((arr_b2[1] & 0xFF) << 8) + (arr_b2[2] & 0xFF) == 17) {
            // case where arr_b2[0] = -x40
            FitDataPackBean fitDataPackBean0 = new FitDataPackBean();
```

During a jump session, **FitData packets are sent for each jump**. They detail how far we are from the planned goal. For example, in Number Count Down mode, the packets provide a status with :

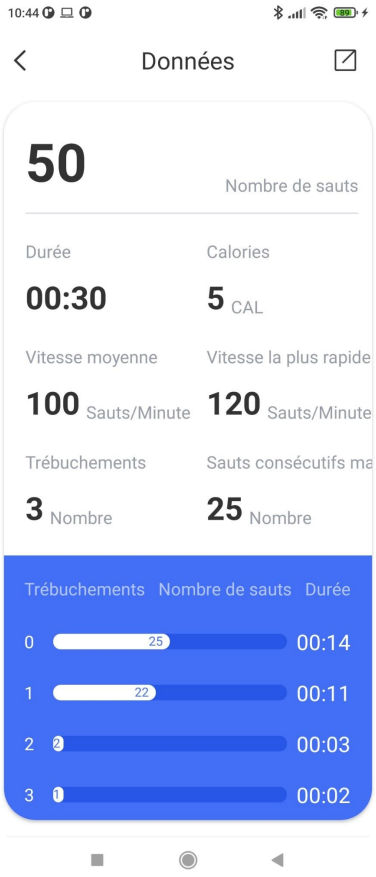
- the number of remaining jumps (called `frequency`)
- the elapsed time (called `timeLength`)
- the last time you stumbled (called `mixRopeTime`)
- and the target number of jumps.

Once the session is complete (or canceled), a **RopeData** packet is sent. This packet summarizes the session and provides the total number of jumps, total elapsed time and a detailed view of the session. Indeed, jump sessions can be seen as a sequence of continuous jumps + corresponding elapsed time. In between each continuous jumps, we *stumble* (which momentarily stops the rope). The RopeData packet provides a table of each continuous jumps/time.

For example, this is the log of a Number Count Down mode session (motion type 3) where I jumped continuously for 25 times in 14 seconds, then stumbled, then 22 times during 11 seconds, then stumbled, then twice for 3 seconds, stumble, then once for 2 seconds:

```
__ropeData=RopeData{id=0, motionType='3', deviceInfoId='0', deviceTypeId='1', costTime='30', avgSpeed='10
```

The RopeData packet does not contain the speeds, deviceInfoId, deviceType: those are computed/provided by the app. The application displays the corresponding screen:



Creating a reliable CTF challenge

The challenge consists in having the participants find out how to initiate a Numbers Count Down mode of 1337 jumps. Participants need to work out the BLE packet to send to the jump rope, and when this is done, we give them a flag. Obviously, the *real* jump rope won't reply with a flag, so my idea was to create a *fake* jump rope (no rope, no jump sensor), which behaves like a real one from a BLE point of view and would additionally serve a flag when the correct command was sent.

When we reverse engineer the rope, we *use* the BLE device and connect to it. Now, it's the opposite: we must *create* a BLE device participants will be able to connect to.

For several weeks (months), I investigated several solutions without success: Bleno, Bluepy, BlueZero, Mirage, direct use of BlueZ and d-bus [BLENO, BLUPY, BZERO, MIRAGE, PUNCH]. While most of these projects *should* have worked in theory, I encountered support issues (e.g. Bleno was abandoned) or bugs (BlueZero, d-bus) or found it was difficult to adapt to my needs (e.g. Mirage is mostly meant to access BLE devices. It can actually clone a BLE peripheral, but then having it react to specific data such as the expected Number Count Down mode requires the implementation of a *scenario*...). To be fair, I am *not* saying these projects do not work, but at that time, I failed to them work whether this was my fault or a bug of the software.

Ph0wn 2021

The deadline for Ph0wn CTF 2021 was approaching, and without any good solution, I decided to create a simili-BLE web server. Participants would go to the web server, see a list of services and characteristics, and they would say which characteristic they wanted to read or write. If they managed to write the expected BLE command, the server would provide the flag in an additional characteristic.

I implemented the web server as a **Flask application**, and dockerized it for the CTF. As much as possible, all challenge servers for the CTF are in Docker containers: this helps restart a faulty service, and also prevents a bug from affecting another challenge.

To prevent a CTF team from "stealing" the flag of another team (i.e one team writes correctly the BLE command but

another team reads the flag), I used Flask *sessions*. Whenever a team would write a BLE command to the expected service/characteristic, the command would be saved in session data. If you tried to read the flag characteristic, the command would be checked before revealing the flag. Any other action (enumerating services, reading...) would erase the session.

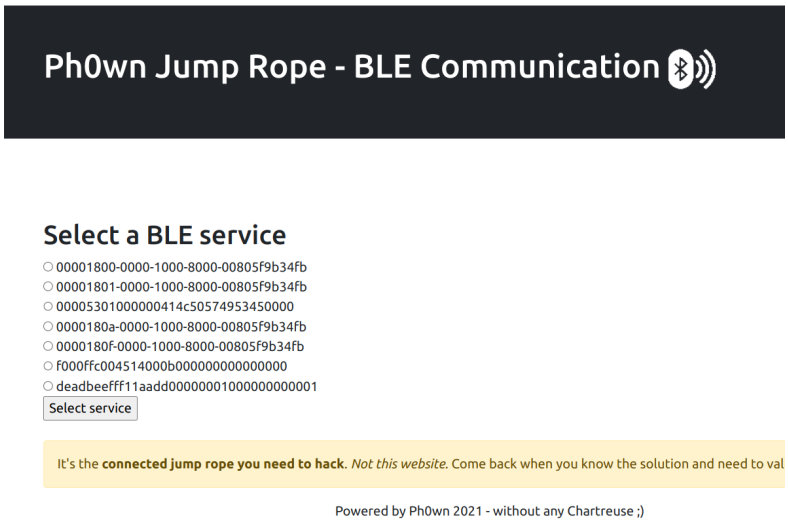


Figure 2: Screenshot of the web server enumerating services. The deadbeef service is added for the CTF. This is where to read the flag when conditions are met.

This solution wasn't totally fool-proof (e.g a team could cause session resets and consequently erase a flag another team should have read), but it more or less worked in the context of Ph0wn where there wouldn't be many teams trying to solve the same challenge exactly at the same time. And we always expect teams to be fair. Note that this assumption may appear *naive*, but in reality, all CTF organizers face the same issue, and at a given point, you have to expect teams to *behave*... It is very difficult to prevent all types of misbehaviors.

Ph0wn 2022

Ph0wn 2021 got cancelled due to COVID-19 2 days before the event. We did the CTF online, but several challenges were not possible online including this Jump Rope challenge. So, we kept it for Ph0wn 2022. This left me more time to improve the validation server. I wasn't satisfied by the web/Flask validation server because it didn't look like a real jump rope and participants didn't have to *send a BLE packet* but to *write hexadecimal value to a web server*.

I participated to a hardware-oriented CTF, *Hardwear.io* CTF, and they offered a BLE challenge/response challenge where we were interacting with a small Arduino-based BLE device. Exactly what I could use for Ph0wn's Jump Rope challenge. So, I ended up discussing with the challenge authors, who were simply using a BLE ESP32 chip and the Arduino ESP32 library.

I decided to use the same solution, and got a few Wemos Lolin32 boards. Like any Arduino device, there is a `setup` function and a `loop` function. The setup does most of the work and creates the services and characteristics to be mimicked, and the read/write callbacks for each characteristic.

Whenever the correct BLE command is received, I write the flag to dedicated (additional) flag characteristic:

```
if (ok == true) {
    pCharFlag->setValue(flag);
} else {
    pCharFlag->setValue(NO_FLAG);

}
```

To ensure a team doesn't steal another team's work, I enforce there is only one single client at any time. I found that starting / stopping BLE advertisment is a reliable way to do this. Actually, this helped me identify a bug in Hardwear.io CTF's code. Connections to their device required reboots. This is because their code was starting advertisment in setup only. Consequently, after a disconnect, advertising wouldn't be restarted, and nobody could

connect unless they had seen the device initially. This is easily solved by starting advertisment periodically in `loop()`.

```
void loop() {
  // enforcing a single client at a time
  if (connected == true) {
    pAdvertising->stop();
  } else {
    pAdvertising->start();
  }
  // wait
  delay(1000);
}
```

Although for this particular challenge we only want *one* connexion at a time, without this enforcement I confirm my devices are perfectly able to handle *several simultaneous BLE connexions*. This is interesting because many Bluetooth-based IoT devices are unable to handle simultaneous connexions... though there is no technical difficulty.

Then, I erase the flag on BLE connections and disconnections:

```
void onConnect(BLEServer *pServer) {
    Serial.println("[+] onConnect: erasing flag");
    pCharFlag->setValue(NO_FLAG);
    connected = true;
}
void onDisconnect(BLEServer *pServer) {
    Serial.println("[+] onDisconnect: erasing flag");
    pCharFlag->setValue(NO_FLAG);
    connected = false;
  }
```

This solution was preferred over the Flask-based server. It is stronger (a CTF team cannot easily reset the Bluetooth session of another team, whereas, with the Flask solution any request to a different characteristic would cause a reset) and nicer (getting the flag requires sending a BLE packet instead of HTTP).

It was successfully used at Ph0wn CTF 2022, where we had 3 different validations servers, for redundancy. Participants could connect to either of them. During the CTF, only one of the servers required a reboot (for an unknown reason).

Conclusion - Take away

I have already reversed several BLE-based IoT devices: a smart toothbrush [TRO18], a smart coffee maker [BMUC20], a connected glucose sensor [PTS20] and others [VB16]. While the precise packets obviously differ, **reversing the Android application has always proved to be a useful** methodology. I recommend the following:

1. Enumerate BLE services & characteristics. Many tools do that (e.g. bluetoothctl, gatttool) but the Android app nRF Connect is my personal favorite.
2. Turn on **notifications** for any characteristic that supports them. This is likely to give you access to many more BLE packets. It was useful to understand live jump data, it was also useful to understand the authorization mechanism of the smart coffee maker.
3. Inspect **Android system logs** (`logcat`) for low hanging fruits: like for the jump rope, you might find all the BLE packets you are looking for.
4. Reverse engineer the **Android application**. In particular, search for UUID such as `1801` (part of the Generic Attribute profile UUID), `2902` (Client Characteristic Configuration descriptor), `2a29` (manufacturer string characteristic). Search also for usage of `android.bluetooth.BluetoothGatt`, `BluetoothGattCharacteristic`, `BluetoothDevice`.

As for deployment in a CTF, or a public demo, special care needs to be taken. While Bluetooth devices are handy at home, or in a small & controlled area, they are a **notable nightmare for any presenter or CTF organizer**. The main issue is BLE connections: people automatically or non intentionally connecting to the device, thus eating up resources of small devices. Compared to WiFi, I encountered less issues caused by distance and walls. Nevertheless, BLE remains a *local* protocol: usually don't expect BLE devices to be seen more than 20 meters away. Some more advice:

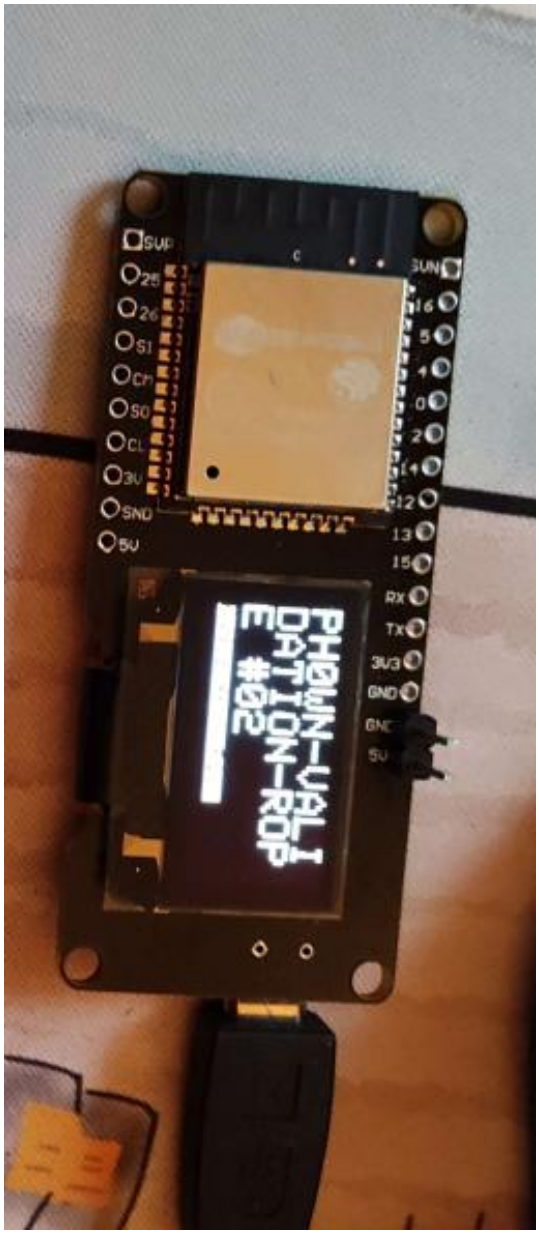


Figure 3: A BLE validation as used during Ph0wn 2022

- Prepare several redundant devices. This will help you cope with BLE connection issues.
- It is generally easier and safer (for CTF) to enforce only 1 connection at a given time.
- If you do need several connections at a same time, make sure the advertiser restarts regularly or you are nearly sure that end-users will complain at some point they can't connect and will need to reboot the device.

References

- [BMUC20] Hacking a Smart Coffee Machine, https://www.youtube.com/watch?v=EvRd3Z41Ff0
- [BLENO] https://github.com/noble/bleno
- [BLUPY] https://github.com/IanHarvey/bluepy
- [BZERO] https://github.com/ukBaz/python-bluezero
- [HALL] https://en.wikipedia.org/wiki/Hall_effect_sensor
- [MIRAGE] https://github.com/RCayre/mirage
- [PTS20] Pique curiosity, not diabetic fingers, https://passthesalt.ubicast.tv/videos/2020-pique-curiosity-not-diabetic-fingers/
- [PUNCH] https://punchthrough.com/creating-a-ble-peripheral-with-bluez
- [RENPHO] https://renpho.com/pages/smart-jump-rope-user-manual
- [TRO18] Is my toothbrush really smart?, Troopers 2018, https://troopers.de/downloads/troopers18/TR18_NGI_BR_Is-my-toothbrush-really-smart.pdf
- [VB16] https://www.virusbulletin.com/conference/vb2016/abstracts/mobile-applications-backdoor-internet-things

Appendix

BLE services

Service UUID	Name / Function
`00001801-0000-1000-8000-00805f9b34fb`	Generic Attribute Profile
`0000180a-0000-1000-8000-00805f9b34fb`	Device Information
`00005301-0000-0041-4c50-574953450000`	Jump Rope Service
`f000ffc0-0451-4000-b000-000000000000`	OTA Service

See `cn.com.heaton.blelibrary.ble.Ble`.

Device Information

- Battery level: UUID=`00002a19-0000-1000-8000-00805f9b34fb`. Handle=0x0029

Jump Rope Service (0x5301)

- Jump Rope Commands: UUID=`00005302-0000-0041-4c50-574953450000`. Handle=`0x0010`. Supports: Write or Write without response.
- Jump Rope Notifications: UUID=`00005303-0000-0041-4c50-574953450000`. Handle=`0x0012`.

To request notifications with `gatttool`:

```
gatttool -b DF:E5:34:0E:42:7D -I
[DF:E5:34:0E:42:7D][LE]> char-read-uuid 00002902-0000-1000-8000-00805f9b34fb
handle: 0x000d   value: 00 00
handle: 0x0013   value: 00 00
handle: 0x002a   value: 00 00
handle: 0x002e   value: 00 00
handle: 0x0032   value: 00 00
[DF:E5:34:0E:42:7D][LE]> char-write-req 0x000d 0100
Error: Characteristic Write Request failed: Internal application error: I/O
```

Then you receive notifications

```
Notification handle = 0x0012 value: ad 04 00 40 00 11 00 00 00 00 00 00 00 00 01 00 00 00 00 00
Notification handle = 0x0012 value: af 04 01 01 5e 46 2f 9a
```

To request notifications with `bluetoothctl`, from menu `gatt`

```
# select-attribute /org/bluez/hci0/dev_DF_E5_34_0E_42_7D/service000e/char0011
[RENPHO-ROPE-R1:/service000e/char0011]# acquire-notify
[CHG] Attribute /org/bluez/hci0/dev_DF_E5_34_0E_42_7D/service000e/char0011 NotifyAcquired: yes
AcquireNotify success: fd 7 MTU 131
```

Motion Type

The mode is also called a *motion type*:

- 01: Number Count Down mode
- 02: Time Count Down mode
- 03: Free Jump mode

In some cases, the used value is actually "motion type - 1", i.e starting at 0

Commands

Commands to initiate a mode begin with `02 00 05`:

Mode	BLE packet	Comment
Numbers Count Down Mode	`02 00 05 81 TT TT TT TT CC CC`	`TT TT TT TT` is the target number of jumps, and `CC CC` is the CRC-16/MODBUS
Time Count Down Mode	`02 00 05 82 TT TT TT TT CC CC`	`TT TT TT TT` is the target number of seconds for the session
Free Jump Mode	`02 00 05 80 00 00 00 00 59 C0`	The CRC is fixed to `59 C0`

Basically, the packet format follows the same pattern with 81 for Number counter down, 82 for Time count down and 80 for free jump.

To stop a given mode: `02 00 05 MT 00 00 00 00 CC CC` where `MT` is the Motion Type and `CC CC` the CRC-16/MODBUS.

Other commands:

Command	BLE packet	Comment
Set Buzzer On	`08 00 01 01 14 C2`	The last 2 bytes are the CRC. They are fixed. When the buzzer is on, the jump makes noise when it is about to start/stop a given mode.
Set Buzzer Off	`08 00 01 00 D4 03`	
Read offline data	`04 02 02 00 00 00 74`	The last 2 bytes are the CRC.
Read offline data size	`07 01 01 A5 CC CC`	
Read offline data status	`A4 01 01 A5 FF E2`	The last 2 bytes are the CRC, fixed to `FF E2` in this case
Clear all offline data	`05 00 01 A5 03 C1`	The last 2 bytes are the CRC, fixed to `03 C1`
Switch to special OTA mode	`06 01 01 A5 47 C1`	The last 2 bytes are the CRC, fixed to `47 C1`
Query BLE Serial number	`03 00 04 00 00 00 00 00 9A 01`	The last 2 bytes are the CRC, fixed to `9A 01`
Set time	`01 00 06 SS mm HH DD MM YY CC CC`	`YY` is current year - 2000, `MM` is month number + 1, `DD` is day, `HH` is hour, `mm` is minutes, `SS` is seconds and `CC CC` is the CRC

Command	BLE packet	Comment
Cancel	020005010000000047FC	

Example of time command for February 10, 2023 at 10:05:44.

```
02-10 10:05:44.144  3409  3441 D TAG     : =2023
02-10 10:05:44.144  3409  3441 D TAG     : =2
02-10 10:05:44.144  3409  3441 D TAG     : =10
02-10 10:05:44.144  3409  3441 D TAG     : =10
02-10 10:05:44.144  3409  3441 D TAG     : =5
02-10 10:05:44.144  3409  3441 D TAG     : =44
02-10 10:05:44.144  3409  3441 E TAG     : HEX=0100062C050A0A021778C8>>>>>>>>>>78C8
```

Jump Rope Notifications

FitData (0x40) packets are sent continuously during a jump session. They follow this format: `40 00 11 DD MM YY TL TL FF FF FF FF MX MX MT TT TT TT TT BB`.

- `DD` is the *day*, `MM` is month and `YY`: year. On my jump rope model, those are not filled and set to `00 00 00`.
- `TL TL`: **current time counter** in seconds (called *time length*). In time count down mode, this decreases from the target count to 0.
- `FF FF FF FF`: **current count of jumps** (called *frequency*). In Number Count down mode, this decreases from the target count to 0.
- `MX MX`: last time (in seconds) a stumble occured (called *mix rope time*).
- `MT`: motion type -1 (i.e number count down mode is 0)
- `TT TT TT TT`: target count set for this session. For example, if we initiated a Number Count Down mode of 1000 jumps, the target count is 1000.
- `BB`: battery percentage.

Status (0x81) with format `81 LL LL [uu [BB [BZ]]] CC CC`. The bytes after `LL LL` are optional.

- `LL LL` indicates how many bytes there are afterwards.
- `uu` is unused
- `BB` indicates battery percentage. If present, the length is >= 2.
- `BZ` is the buzzer status. If present, the length is >= 3. If not present, buffer status is set to -1 (unknown).
- `CC CC` is the CRC.

Example: A BLE packet `810003004601eeed` provides status data (0x81). The length of the payload data is 3 bytes (`00 03`). Battery percentage is `0x46` = 70%, and buzzer status is on (`01`). Bytes `EE ED` are the CRC.

```
02-06 17:22:49.086  8148 11181 D TAG     :  --> with buzzer
```

RopeData (0xA0) packets follow this format: `A0 SS SS MM MM MM MM uu uu MT CT CT JJ JJ MX MX RD RD... CC CC`. They provide data on a sequence of jumps.

- `SS SS`: number of bytes of the RopeData (not counting `A0 SS SS` and the trailing CRC).
- `MM MM MM MM`: not clear. a time in time millis
- `uu uu`: not used
- `MT`: motion type -1
- `CT CT`: **total number of elapsed seconds** (called *cost time*)
- `JJ JJ`: **total number of jumps**
- `MX MX`: **number of stumbles**
- `RD RD ..`: details of the sequence of jumps of the session. (called *rope mixing data*). Each entry in the sequence contains a current jump counter and a current elapsed seconds.
- `CC CC`: CRC

The *Rope Mixing Data* is a sequence of entries with the format `CJ CJ LT LT` where `CJ CJ` is an integer representing the current value of the jump counter, and `LT LT` is an integer presenting the current elapsed seconds. In the sequence, both the counter and the elapsed seconds increment. So, for example, if we have the following sequence `AA AA BB BB CC CC DD DD`, this provides data for 2 series of jumps, where the first one contained `AA AA` number of jumps during `BB BB` seconds, and the second series contained `CC CC` - `AA AA` jumps during `DD DD` - `BB BB` seconds.

OfflineData (0x84) packets with format `84 00 02 xx xx CC CC` indicate offline data has been sent. In some other cases, OfflineData packets simply provide the same as a RopeData, except they are sent "offline" (outside a session).

```
02-10 10:20:20.519  3409  6826 D TAG     : command=132
02-10 10:20:20.519  3409  6826 I TAG     : =8400190000001e000001001e003200030019000e002f00190031001c
...
02-10 10:20:20.519  3409  6826 I TAG     : __ropeData=RopeData{id=0, motionType='3', deviceInfoId='0', dev
```

Reset BLE (0x86) which start with `86` force the app to disconnect and start re-scanning.

Buzzer Data (0x88) with format `88 uu uu BZ` provide the current buzzer status of the rope:

- `uu` is unused
- `BZ` is the current buzzer status: 0 (off) or 1 (on)

Example of Memory Group

The following packets belong to the same memory group `01`:

```
ad0100400011000000002500000002001900000 0
af010103e864c222
```

There are only 2 packets in this group: the first one (`AD`) and the last one (`AF`). The payload is `400011000000002500000002001900000 0` + `03e864c222`. The last 2 bytes `C2 22` are the CRC of the payload. As the packet begins with `40`, this is a FitData packet.

Example of RopeData packet

```
ad0500a0001d000000560000000057000500040 0
af05010100190002002b0003002e00050037abb5
```

These 2 packets from the same group form a packet with payload `a0001d000000560000000057000500040001001900020 02b0003002`

- A0 indicates this is *Rope Data* packet
- RopeData size: 0x1d (29 bytes). This does not count `A0 00 1D` nor the CRC `AB B5`
- Motion Type -1 is `00`, so this is Number Count Down Mode.
- Cost Time is `00 57`, i.e 87 seconds.
- Number of jumps is `00 05` : 5 jumps
- Number of stumbles is `00 04`

Then, we have the stumble details:

- 1 jump during 0x19=25 seconds `00 01 00 19`
- Another jump (current jump amount=2) during 18 seconds (last time is 0x2b=43 seconds) `00 02 00 2b`
- Another jump (current jump amount=3) during 3 seconds (last time 0x2e=46 seconds) `00 03 00 2e`
- 2 jumps (current jump amount=5) during 9 seconds (last time 0x37=55 seconds) `00 05 00 37`

```
02-06 17:24:06.285  8148 11181 D TAG     : maxJumpList=[1, 1, 1, 2, 0]
```

Finally, the packet ends with the CRC: `AB B5`. This is CRC-16/MODBUS over `a0001d000000560000000057000500040001001900 0`

From this data, the app computes additional data:

- Average speed: nb of jumps / (total time/60) = 5 / (87/60) = 3.
- Max continuous jumps: 2

```
02-06 17:24:06.285  8148 11181 I TAG     : __ropeData=RopeData{id=0, motionType='1', deviceInfoId='0', dev
```

Example of FitData packet

```
ad0100400011000000002500000002001900000 0
af010103e864c222
```

These 2 packets from the same group form a packet with payload `400011000000002500000002001900000003e864c222`.

- `40` indicates this is a FitData packet
- `00 00 00`: day, month and year are not filled

- `00 25` corresponds to 37 seconds (0x25)
- `00 00 00 02` corresponds to frequency of 2
- `00 19` corresponds to 25 seconds of mix rope time (0x19)
- `00` indicates Number Count Down mode
- `00 00 03 e8` indicates a target number of jumps of 1000
- `64` indicates 100% (0x64) battery
- `C2 22` is the CRC

```
02-06 17:23:17.328  8148 11181 I TAG     : =FitDataPackBean{day=0, month=0, year=0, timeLength=37, frequen
```