|  |
| --- |
| Close-up image showing the leaf-sides of two oversized books side-by-side on a bookshelf, with additional books in soft focus background |
| IOT REDBOOK  BLUETOOTH LOW ENERGY |
| |  |  |  | | --- | --- | --- | | Navneet Mishra, Jacob Victor |  |  | |

**Table of Contents**

[Bluetooth Low Energy 2](#_Toc535425392)

[Central device and peripheral device 2](#_Toc535425393)

[Broadcaster and observer 3](#_Toc535425394)

[BLE Stack 4](#_Toc535425395)

[GAP 4](#_Toc535425396)

[Advertising Process 4](#_Toc535425397)

[GATT 5](#_Toc535425398)

[GATT Transactions 5](#_Toc535425399)

[Services and Characteristics 5](#_Toc535425400)

[Services 5](#_Toc535425401)

[Characteristics 6](#_Toc535425402)

[BLE Tools 7](#_Toc535425403)

[Ubertooth One 7](#_Toc535425404)

[CSR Bluetooth Adapter 7](#_Toc535425405)

[Crackle 7](#_Toc535425406)

[GATTacker 8](#_Toc535425407)

[Blue Hydra 8](#_Toc535425408)

[Ubertooth Utils 8](#_Toc535425409)

[Wireshark 8](#_Toc535425410)

[How to? 8](#_Toc535425411)

# Bluetooth Low Energy

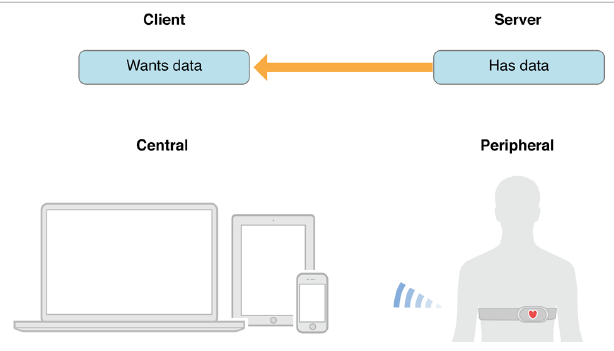
BLE or is one of the most common platforms found in a number of smart devices ranging from smart homes to medical device utilities to even fitness trackers and wearables. One of the reasons for the growing popularity of BLE is that pretty much all smartphones that we use today support BLE, thus making it easier to interact with BLE-based IoT devices.

BLE is designed for devices with resource and power constraints which BLE effectively solves by providing short bursts of long range radio connections, thus significantly saving battery consumption. BLE was initially introduced in Bluetooth 4.0 specifications focusing on the devices which needed a mode of communication with extreme low power consumption.

A BLE device can operate in four different modes based on its current connection and operational phase:

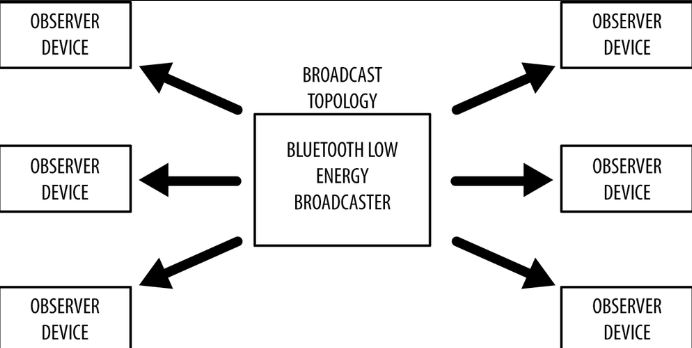
# Central device and peripheral device

* **Peripheral** devices are small, low power, resource constrained devices that can connect to a much more powerful central device. Peripheral devices are things like a heart rate monitor, a BLE enabled proximity tag, etc. Peripheral devices advertise themselves.
* **Central** devices are usually the mobile phone or tablet that you connect to with far more processing power and memory. They scan for advertisement packets and initiate connections.

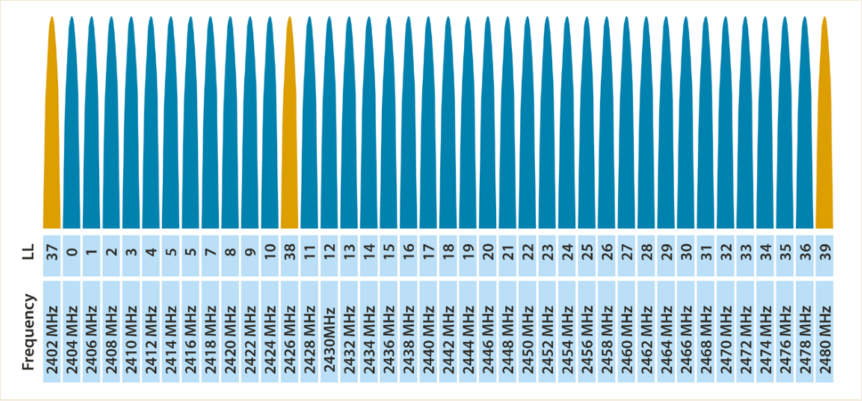


# Broadcaster and observer

As the name implies, a broadcaster is a device that broadcasts data, whereas an observer is a device which scans for advertisement packets. However, the major difference here compared to the previous classification type is that the broadcaster is non connectable and the observer can't initiate connections. An example of this would be a weather station which emits temperature data continuously acting as a broadcaster, whereas a display that receives the broadcasts and shows it on the screen is an observer.



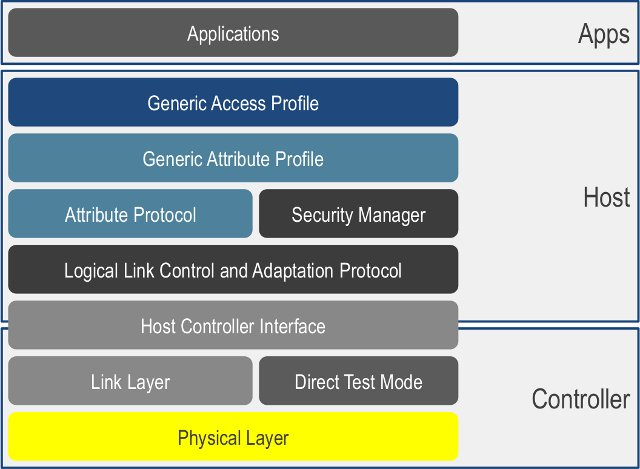
BLE consists of 40 different channels—3 advertisement and 37 data channels, as shown in the following image



[Image Source](http://www.connectblue.com/press/articles/shaping-the-wireless-future-with-low-energy-applications-and-systems/)

BLE also performs frequency hopping spread spectrum, which means that it keeps changing channels on every event. However, the tools that we are going to use in the coming sections will be able to follow a device via the hops and be able to sniff the data for the BLE communication.

# BLE Stack



[Image source](https://www.cnx-software.com/wp-content/uploads/2013/06/Bluetooth_4.0_Stack_Architecture.jpg)

## GAP

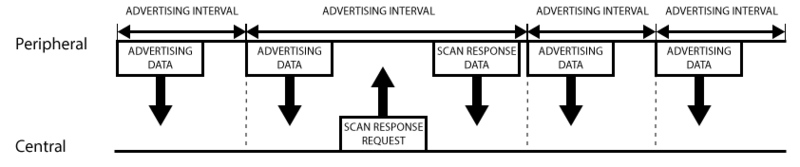
It controls connections and advertising in Bluetooth. GAP is what makes your device visible to the outside world, and determines how two devices can (or can't) interact with each other. There are two ways to send advertising out with GAP.

* The *Advertising Data* payload - advertising data payload is mandatory, since this is the payload that will be constantly transmitted out from the device to let central devices in range know that it exists.
* The *Scan Response* payload - The scan response payload is an optional secondary payload that central devices can request, and allows device designers to fit a bit more information in the advertising payload such a strings for a device name, etc.

### Advertising Process

A peripheral will set a specific advertising interval, and every time this interval passes, it will retransmit its main advertising packet. Longer delays save power but feel less responsive if the device only advertises itself once every 2 seconds instead of every 20ms.If a listening device is interested in the scan response payload (and it is available on the peripheral) it can optionally request the scan response payload, and the peripheral will respond with the additional data.

The peripheral stops advertising as and when the connection is established



## GATT

It defines the way that two Bluetooth Low Energy devices transfer data back and forth using concepts called **Services** and **Characteristics**. It makes use of a generic data protocol called the **Attribute Protocol (ATT)**, which is used to store Services, Characteristics and related data in a simple lookup table using 16-bit IDs for each entry in the table.

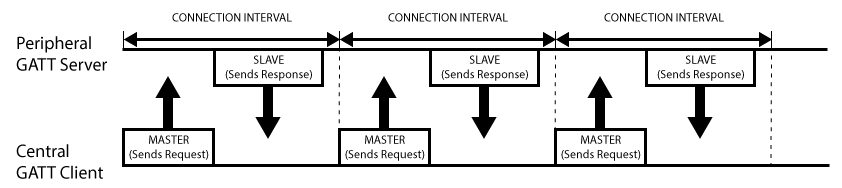
GATT comes into play once a dedicated connection is established between two devices, meaning that you have already gone through the advertising process governed by GAP.

### GATT Transactions

An important concept to understand with GATT is the server/client relationship. The peripheral is known as the **GATT Server**, which holds the ATT lookup data and service and characteristic definitions, and the **GATT Client** (the phone/tablet), which sends requests to this server. All transactions are started by the master device, the GATT Client, which receives response from the slave device, the GATT Server.

When establishing a connection, the peripheral will suggest a 'Connection Interval' to the central device, and the central device will try to reconnect every connection interval to see if any new data is available, etc. It's important to keep in mind that this connection interval is really just a suggestion, though! Your central device may not be able to honour the request because it's busy talking to another peripheral or the required system resources just aren't available.

The following diagram should illustrate to data exchange process between a peripheral (the GATT Server) and a central device (the GATT Client), with the master device initiating every transaction:



## Services and Characteristics

GATT transactions in BLE are based on high-level, nested objects called **Profiles**, **Services** and **Characteristics.** Profiles

A Profile doesn't actually exist on the BLE peripheral itself, it's simple a pre-defined collection of Services that has been compiled by either the Bluetooth SIG or by the peripheral designers. The [Heart Rate Profile](https://developer.bluetooth.org/TechnologyOverview/Pages/HRP.aspx), for example, combines the Heart Rate Service and the Device Information Service.

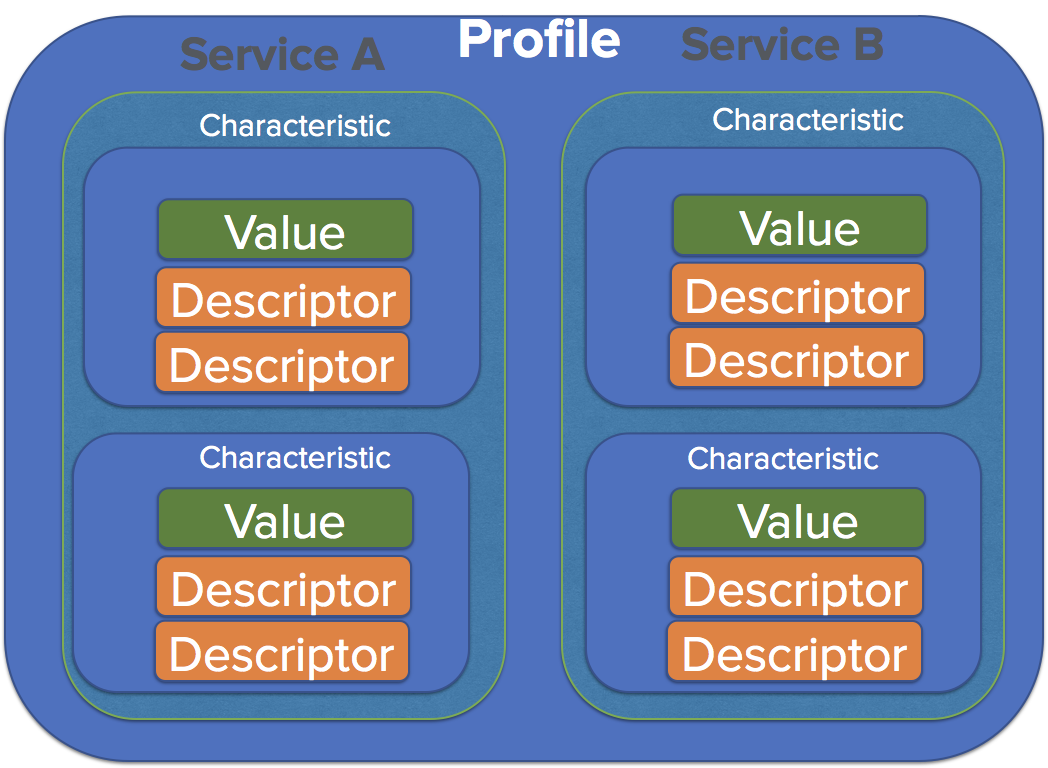
### Services

Services are used to break data up into logic entities, and contain specific chunks of data called characteristics. A service can have one or more characteristics, and each service distinguishes itself from other services by means of a unique numeric ID called a UUID, which can be either 16-bit (for officially adopted BLE Services) or 128-bit (for custom services).

### Characteristics

The lowest level concept in GATT transactions is the Characteristic, which encapsulates a single data point (though it may contain an array of related data, such as X/Y/Z values from a 3-axis accelerometer, etc.).

Similar to Services, each Characteristic distinguishes itself via a pre-defined 16-bit or 128-bit UUID. Characteristics are the main point that you will interact with your BLE peripheral, so it's important to understand the concept. They are also used to send data back to the BLE peripheral, since you are also able to write to characteristic.



As we can see from the preceding diagram, the overall data is organized into the lowest elements of Characteristic, which hold a Value and Descriptor. An example of this would be heart-beats-per-minute and its value being stored in characteristic.

Next, various similar characteristics are enclosed in services. An example of a service would be heart-rate-service, which contains various characteristics such as heart-beats-per-minute, irregular-heart-beats, and panic-attacks. Services also have a 16-bit UUID using which they can be referred from to the Bluetooth SIG database of Services UUID available [here](https://www.bluetooth.com/specifications/gatt/services).

Next, this entire service is enclosed within a Profile, which could be a generic profile, in this case, something like a heart health profile, which contains various services such as heart-rate-service and heart-oxygen-service.

As mentioned earlier, our goal during sniffing is to find the value of characteristics, which are being read and written. These characteristics are usually referred to as handles, which is what we will see once we capture the traffic.

Moving on, the other important component of the BLE stack is L2CAP. L2CAP stands for Logical Link Control and Adaption Protocol and is responsible for primarily taking data from the other layers and encapsulating the data in a proper BLE packet structure.

# BLE Tools

## Ubertooth One

* Bluetooth Sniffer and injector
* 2.4 GHz transmit and receive capabilities
* Open source
* Easily integrates with Wireshark

To sniff BLE traffic effectively, it is important to identify the device which could be advertising on any of the three advertising channels. For this, it is important to have a setup of three Ubertooths rather than one.



## CSR Bluetooth Adapter

* Bluetooth Adapter
* 2.4 GHz transmit and receive capabilities
* Supports BLE 4.0
* Freely Available on online shopping sites
* Easily integrates with Wireshark



## Crackle

* Cracks BLE Encryption
* Open source
* Website: http://lacklustre.net/projects/crackle/

As per their website, it can guess or very quickly brute force TK, with TK it derives all other keys

Crack the LTK

$ crackle -i <pairing-capture.pcap> -o <output.pcap>

Decrypt the encrypted application communication using the cracked LTK

$crackle -i <encrypted-ble-comm.pcap> -o <decrypted-ble-comm.pcap> -l *“LTK”*

## GATTacker

* GATTacker is a BLE MiTM tool.
* It needs two machine to act as Bluetooth proxy.
* One acts as a device and pairs with the phone and other acts as the app which connects to the device.
* You need two BLE hardware to perform this.
* This is based on Node.

## Blue Hydra

To perform initial recon on the BLE devices ([Download link](https://github.com/pwnieexpress/blue_hydra))

## Ubertooth Utils

To perform sniffing and packet capture for our BLE devices ([Download link](https://github.com/greatscottgadgets/ubertooth))

## Wireshark

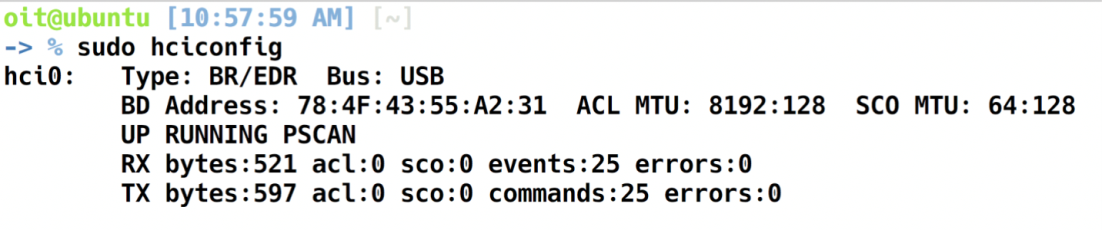
Packet analysis tool, also compatible with BLE ([Download link](https://www.wireshark.org/download.html))

# How to?

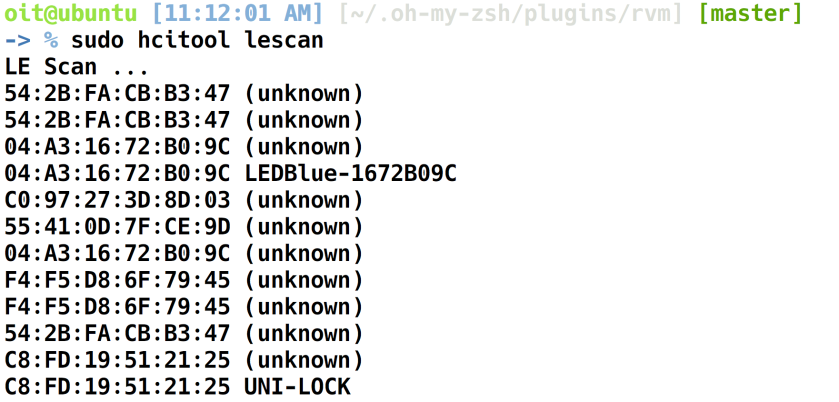
When working with BLE, our methodology is to first find out the target device's address and sniff the traffic for that specific address while performing an operation with the target BLE device.

This will allow us to find specific BLE handles that are being written on the device to perform a certain action. To give a better insight to what BLE handles are, they are simply a reference to the various properties a BLE characteristic has.

1. Ensure that the Bluetooth Adapter dongle is connected to your virtual machine and you are able to see the hci interface, as shown in the following screenshot:



2. The first thing that we will do in order to interact with BLE devices around us is to look at all the devices around us and find their Bluetooth addresses. This uses the lescan (Low Energy Scan) functionality of Hcitool to look for all the BLE advertisements in the vicinity, as shown in the following screenshot:



As you can see, we are able to identify a number of devices around us along with their addresses. Next, we can use Ubertooth to sniff the traffic for a given device as shown next. For us as security researchers and penetration testers, it can help identify security issues such as clear-text data transmission and also identifying which handles are being written and read during a network communication.

1. To use Ubertooth to sniff the connections from a given device following it, use the following syntax:

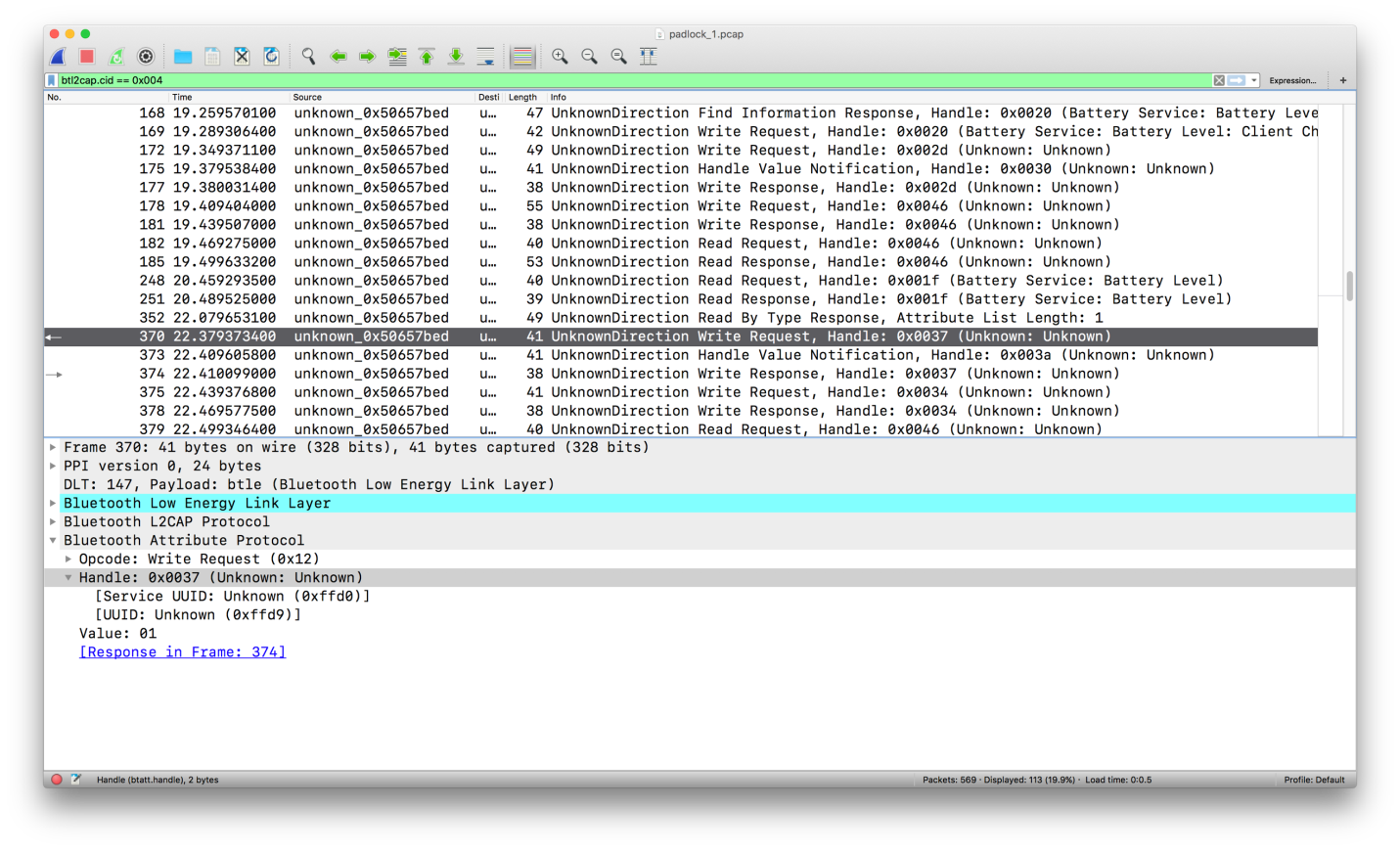
sudo ubertooth-btle -f -t [address] -c [capture-output]

Replace [address] with the address of the device, which we identified in the previous step.

The [capture-output] could either be a file or a pipe, in case we want to do an active traffic interception.

Let's use /tmp/pipe as a capture interface with one end of the pipe getting input data from the Ubertooth and the other end of the pipe showing the data in Wireshark.

* To do this, open up another terminal window and type in mkfifo /tmp/pipe. Once done, go to Wireshark | Capture Interfaces | Manage Interface | New Interface | Pipes and add the value /tmp/pipe and save the interface.
* Next, start sniffing on the /tmp/pipe interface in Wireshark, which you have just created. Depending on the action you are performing and your target device, you will be able to see the BLE traffic show up in Wireshark, as shown in the following screenshot:



We have also applied a filter in the preceding screenshot of **btl2cap.cid==0x004** to ensure that we get the packets which have useful data. As you can also see in the image, we have got a number of read/write requests, along with details of the handles and the value that was written to that specific handle. In this case, the handle 0x0037 and the value 1 correspond to unlocking of a BLE-based smart lock.

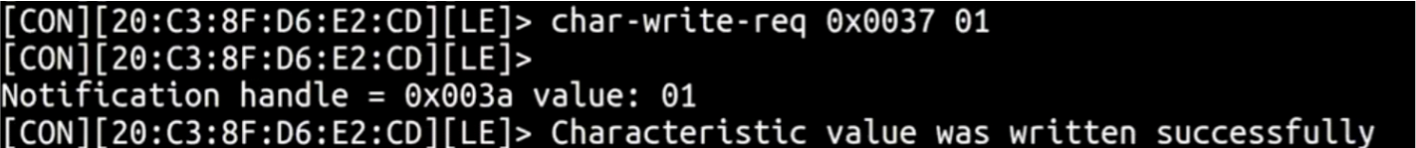
Now that we know what value is being written to the handle in order to perform a specific action, we can write that particular handle ourselves, without the need for Ubertooth. For this, we will use the BLE adapter dongle and a utility called gatttool.

1. To do this, launch up gatttool, as shown next, along with the -b and -I flags providing the Bluetooth address and specifying it open in interactive mode:

sudo gatttool -b [Bluetooth-address] -I

[gatttool prompt] connect

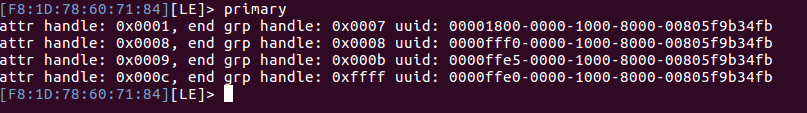
2. Next, all we need to do here is send a write request to the target device specifying the value that we want to write to the handle, as shown in the following screenshot:



This has unlocked the smart lock since the smart lock detected that the handle 0x0037 now has the value of 01, which is related to the state of smart lock being unlocked.

This is how you can interact with a BLE-based IoT device, figure out which handles are being written and then write those handles yourself.

You can also look at other properties of the device by looking at all the values of services, as shown in the following screenshot:



This can also be done for characteristics, as shown in the following screenshot:

