CHAMPLAIN COLLEGE | LCDi Leahy Center for Digital Investigation

175 Lakeside Ave, Room 300A
Burlington, Vermont 05401
Phone: (802)865-5744
Fax: (802)865-6446
http://www.lcdi.champlain.edu

04/13/2018

# Bluetooth Tracking

## Contents

# Introduction

This project builds off some of the tools and methodologies focused on in the Bluetooth Tracking Project of Fall 2017. That report is available from the Leahy Center of Digital Investigation for any interested individuals. To summarize, last semester's project focused on using Bluetooth technologies Ubertooth and BlueHydra to trilaterate the location of specific Bluetooth devices. The same tools can be used in a larger scale tracking effort that focuses on using a mesh network of nodes to track a Bluetooth signal across a wide geographical area. A robust backend database will enable the data correlation portion of this project.

## Background

Our methodology and toolset was based on the previous reporting by the LCDI team in the Fall of 2017. That team focused on the same technologies and protocols our team used to create a new tool set. That project had a markedly different objective, but the tools used are the same. This project was developed based on the possibilities imaged during that project that did not fit with the previous objectives.

This type of latent tracking most closely resembled the reported RFID based surveillance of EZ-Pass devices, as reported by the ACLU in New York City in 2015 that function as part of a traffic management system. The New York State Department of Transportation has also implemented this system outside of NYC. By laying out nodes in a defined pattern across a large area, it is possible to identify how traffic moves through an area as well as if specific devices have been seen in an area. This is an attempt to replicate similar results using open source projects and low cost hardware.

## Purpose and Scope

This research will address the possibilities presented by creating a distributed network of Bluetooth nodes that constantly monitor Bluetooth packets and record the devices they encounter. This project will not focus on defining the specific location of the device, as it would require an unreasonable density of nodes. The scope of this project includes scaling the proposed node network to include three nodes across a multi-storied building.

### Research Questions

1. Can a network of relatively low cost nodes be used to track a specific Bluetooth device ID across a geographical area?
2. How will the planned distribution of nodes scale up over the expanse of a building?
3. How can we implement Elasticsearch and Kibana to organize our data?

## Terminology

**2.4 GHz -** The frequency range from 2.4 to 2.4835 GHz, which is unlicensed and used for many communications applications, including Bluetooth and Wi-Fi (PC Mag, 2017).

**Bluetooth** - A low-power wireless connectivity technology used to stream audio, transfer data and broadcast information between devices (Bluetooth.com, 2017).

**Blue Hydra** - A Bluetooth device discovery program built on top of the bluez library. BlueHydra makes use of Ubertooth where available and attempts to track both classic and low energy (LE) Bluetooth devices over time (PwnieExpress, 2017). Also referenced as blue_hydra.

**Elasticsearch -** A open source, RESTful search engine built on top of Apache Lucene. It is Java based and can search and index document files in diverse formats. This is developed by elastic. (Elasticsearch)

**GeoJSON -** A specialized type of JSON file that is used for storing geographic coordinates used in points, lines, and polygons. (GeoJSON)

**Kibana -** A powerful database visualization tool which is used with Elasticsearch to collect statistics and show collected data through graphs.

**LAP** - Lower Address Portion, lower segment of the MAC address used for Bluetooth Packet addressing (mxs).

**NAP -** Non-significant Address Portion, assigned to manufacturer (mxs).

**Node** - A Raspberry Pi that has been outfitted with an Ubertooth, wireless dongle, and Bluetooth dongle. Each node runs a copy of our script and sends data to our central database.

**Python** - Python is a programming language that lets you work quickly and integrate systems more effectively (Python, 2017).

**Raspberry Pi** - A low cost mini-computer that is capable of running a lightweight Linux distribution. These devices are very portable and allow them to be placed anywhere with a power source and wireless network connection.

**TeamViewer** - A free remote management software installed on each node that allows us to manage each node and run commands via the open web. Nodes enabled with this software would be able to automatically beacon “home” to the teamviewer servers. Using the website GUI, we were able to monitor which nodes were up and connect to them regardless of what network the node was connected to.

**UAP** - Upper address Portion of the MAC, 1 byte in size, links NAP and LAP for full address (mxs).

**Ubertooth -** Project Ubertooth is an open source wireless development platform suitable for Bluetooth experimentation. Ubertooth ships with a capable BLE (Bluetooth Smart) sniffer and can sniff some data from Basic Rate (BR) Bluetooth Classic connections (Greatscottgadgets, 2017).

**Virtual Machine / VM -** Emulation of software, commonly an operating system, on hardware that is not physically dedicated to the emulated software itself.

# Methodology and Methods

Our data will be collected using individual nodes, each of which is comprised of a Raspberry Pi 3, an Ubertooth One, a dedicated Bluetooth Adapter, and a dedicated Wi-Fi USB card. Each node will collect data using our custom script and Blue Hydra. Collected data will be sent to the Elasticsearch instances hosted on a local virtualized environment. The data can later be analyzed using Kibana, a visualization tool also created by elastic, to track specific Bluetooth addresses, or develop statistics about Bluetooth traffic seen by specific nodes.

## Equipment Used

Each sensor requires several specific pieces of hardware. To perform data collection, we used the Raspberry Pi microcomputers with SD cards as the hardware base. We then connected several peripheral devices to the Raspberry Pi; an Ubertooth One, to detect non-discoverable devices, a dedicated Bluetooth adapter to detect discoverable Bluetooth Devices, and a dedicated Wi-Fi adapter to reliably connect to wireless networks.

Table 1: Components of each node

| Device | OS Version | Comments |
|---|---|---|
| Raspberry Pi | Raspbian Stretch Lite | Hardware for individual nodes |
| Ubertooth One | Ubertooth 2017-03-R2 | Sniffs non-discoverable Bluetooth packets |
| Kinivo BTD-400 | N/A | Dedicated Bluetooth Adapter |
| CanaKit WiFi Adapter | N/A | Dedicated WiFi Adapter |

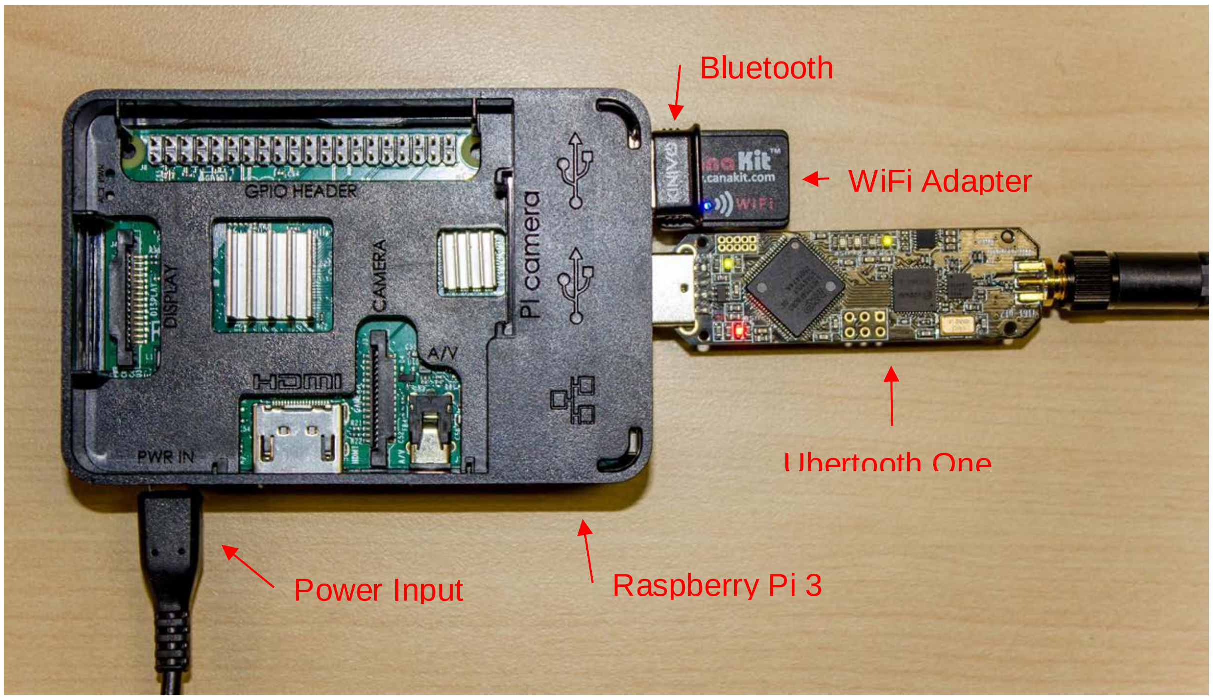


Figure 1: (Components of each node)

## Node Provisioning

We chose to use the Raspbian Stretch Lite image on our Raspberry Pis. This operating system is flashed on to a micro SD card, a process for which two pieces of software are required. The first software is the operating system, which can be downloaded from the official Raspberry Pi website. The other software is for writing the image to the SD card. We used Etcher to accomplish this. Any SD card flashing software can be used for this process. After the card was imaged with the appropriate version of the operating system and inserted in to the Raspberry Pi, the default setup procedure of the operating system can be followed.

After the operating system is installed and configured, a few libraries needed to be installed on the Pi for our custom script to function. The two most prominent of these are Blue Hydra by PwnieExpress and Ubertooth tools by Great Scott Gadgets. These are the specific tool sets used, and they require a number of dependencies to function properly. This software and all the other prerequisites for our script can be installed using the setup file found in Appendix A. This will install the dependencies required for the data collection script itself, and make sure those tools are in the correct folder to be accessed. We highly recommend using the setup script to provision the hardware, as the scanning script will not function if packages are installed in specific locations. Please note the Pi requires internet access in order to download the packages. Ethernet or wireless will both serve this purpose.

While Ethernet is the easier option to configure initially, wireless connections are the preferred option for the most flexible node location. To connect our nodes to the LCDI wireless network, we used dedicated wireless adapters after running into connectivity issues with our on-board wireless cards. Doing this is relatively simple. We followed the instructions laid out in a tutorial that edits the content of the */etc/network/interfaces* file to reflect the network SSID and password for the wireless network we were on (Adafruit, 2018). There were some settings in this file that varied for us due to the specific hidden network we were connecting to, but it was much simpler when connecting to open networks for testing.

When the setup file has finished running the Pi will restart. Blue Hydra can be started using the command: "./home/pi/installations/blue_hydra/bin/blue_hydra". This creates a log file (named blue_hydra.yml). Then, edit this log file using the command "nano /home/pi/installations/blue_hydra/blue_hydra.yml" to reflect the following changes.

- Set the option **log_level** to **debug**
- Set the option **rssi_log** to **true**
- Set the option **aggressive_rssi** to **true**
- Set the option **info_scan_rate** to **5**

After making these changes press 'Ctrl-X' to exit then 'Y' and enter to save the changes you made. When all this is completed the device will be setup and ready to run the data collection script. This script can be found in Appendix B.

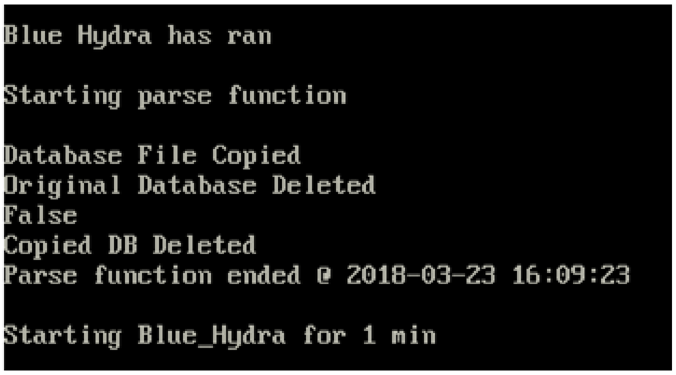
To add external access using TeamViewer first download the file from this link. Then move the files to your Pi using either FTP or a flash drive. Once the file is on your Pi navigate to where it is located and run the command *sudo dpkg -i teamviewer**. If there are dependency errors then run *sudo apt-get install -f*. Finally, run the command *sudo teamviewer setup* and follow the on screen instructions.

## Data Collection

The data collection functionality of this setup is outlined in the Python Script named 'SpyPies.' In this script, Blue Hydra runs for one minute, collecting data that is written to a database file. At the end of the minute, the script executes a check to ensure that the database contains information. If the database is empty, the script will continue running Blue Hydra in one minute increments until data is written to the database.

When data is output to the database, the script stops Blue Hydra and copies the original database to a temporary database. This stop, copy, start, and continue keeps the script from performing commands against a database that is being dynamically updated. The temporary database is where the script will parse out the seen devices in order to begin the update process. The original database is deleted or "cleared" in order to prevent duplications in the copy-over process. The script pulls each line of the database individually, appends the hostname of the specific Raspberry Pi, and sends it to the Elasticsearch server. Once the data is in our Elasticsearch server, we can interpret it using Kibana, to present a visualization. Once the copied database has been completely processed, it will also be deleted.

While the send to server process is running, Blue Hydra is running in the background, still collecting data and recreating the original database with new data. This process will repeat in a cycle until the script is manually stopped or crashes from a generated error.



```
Blue Hydra has ran

Starting parse function

Database File Copied
Original Database Deleted
False
Copied DB Deleted
Parse function ended @ 2018-03-23 16:09:23

Starting Blue_Hydra for 1 min
```

Figure 2: (The running script)

## Server Provisioning

For this project, we implemented a central VM server running Ubuntu 16.04.3 LTS with Elasticsearch and Kibana. Elasticsearch was used for data collection and searching, and Kibana was used through a web interface to visualize our data in a more user-friendly format. We ran this server on the LCDI's internal network, which suited our purposes for testing in the Lakeside building. Raspberry Pi nodes were configured and connected to the network, acting as the Bluetooth sensors. They sent data to the server that was then sorted and could be visualized through the use of various graphs and charts.

### Configuring Elasticsearch

We implemented Elasticsearch because of its flexibility and the availability features that it offers. This was the easiest way to store our data and allows us to easily visualize it with Kibana. Elasticsearch uses nodes to operate, which interconnect with each other, and would allow for the project to easily expand when another node is needed. Most of the steps followed for the setup process were straight forward and were provided from the Elasticsearch Installation page. There are a few requirements such as Java and the Java Development Kit, but the rest of the process is simple enough to get running quickly. We chose to use an Ubuntu Server instance with 1 vCPU and 4GB of RAM, the minimum amount required to run an Elasticsearch server in production.

## Configuring Kibana

Kibana is a powerful tool that allows for a host of interactions with data from Elasticsearch. Kibana requires an active connection to Elasticsearch to run properly and find data. Once data is present in Elasticsearch, an index can be selected for visualization. In the context of our project, coordinate and region maps were most appropriate. Unfortunately, for an area as small as our testing environment, Kibana did not offer visualizations as geographically precise as we needed. Adding our own arbitrary building maps was also an idea that was explored, but was not offered by Kibana. We attempted creating our own region map that corresponded to the physical location of the Lakeside building of Champlain College, but did not have enough time to complete all the necessary steps. If a custom region map is desired, it must correspond to physical geographic coordinates that are determined by precise latitude and longitude.

## Custom Region Maps

To create custom maps, a geojson file must be created and imported into the server running Kibana. This file can be created with any simple geojson editor, with many options available through a quick web search. We chose http://geojson.io/ to create ours. Once the geojson file is imported into the server, it must be served to Kibana through a web server. In our case, we ran a web server using Apache on our Ubuntu VM alongside Kibana and Elasticsearch. After this was in place, the location of the custom region map needed to be specified in the Kibana configuration file, *kibana.yml*. An official blog post on Elastic's website assisted us greatly with all of these steps, and finally got us to the point of seeing our own geojson file within Kibana (Walkom, 2018). In order to increase the maximum zoom level, X-Pack needs to be installed on the server, and can be obtained with a free basic license offered on the Elastic website.

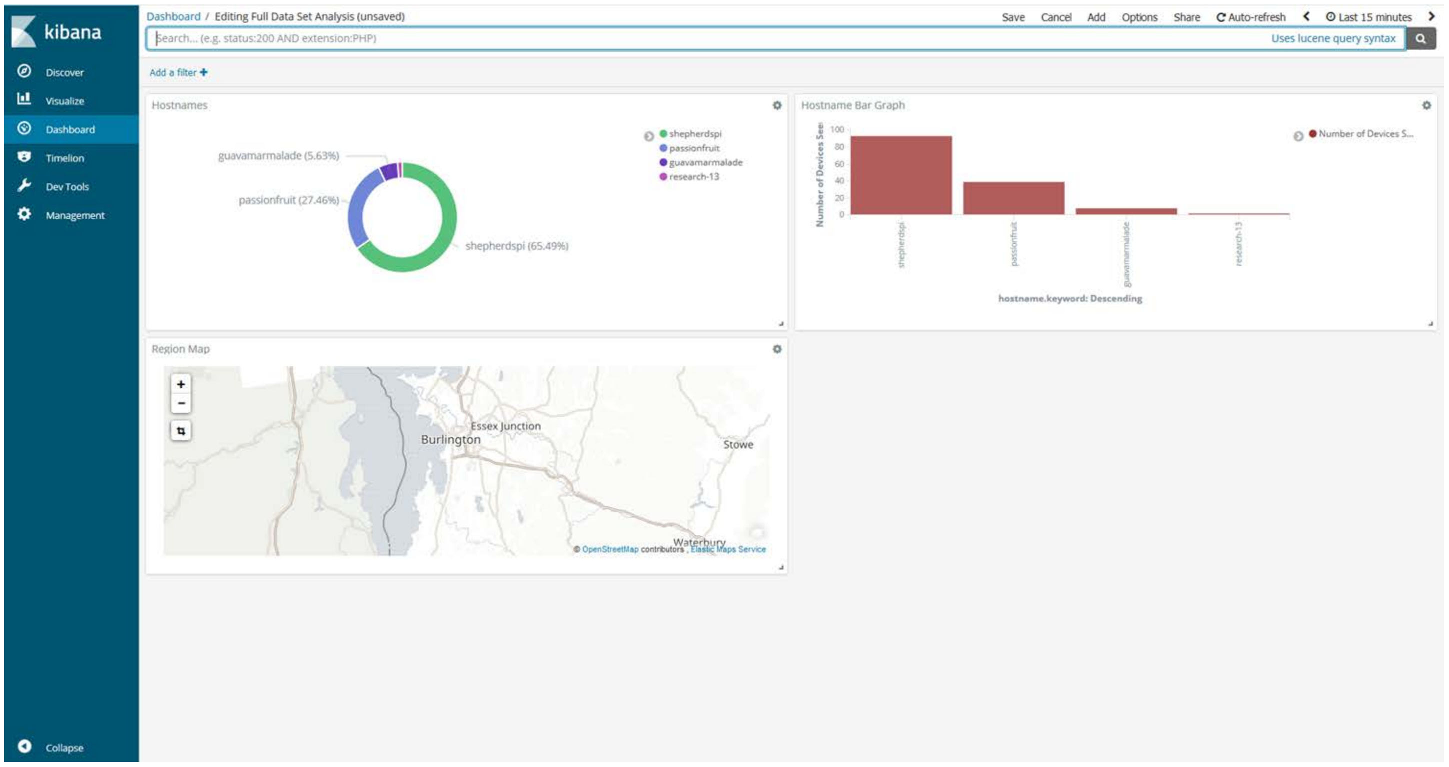


Figure 3: (Our Kibana Dashboard)

## Analysis

We tested out configuration using the local LCDI Wi-Fi in order to keep testing within a controlled environment. This allowed for the quickest and most cost effective solution while still allowing a reasonable geographic distance between Raspberry Pi nodes. Each was placed in a different area of the building and connected to the LCDI wireless network. We ensured they were online using the TeamViewer setup outlined above. This enabled us to tell if a node wasn't transmitting data because it wasn't connected or if the issues stemmed from other functions. This scenario enabled us to test the underlying theory of how we expected the ecosystem to work. It also revealed issues with wireless connectivity. In some cases, it appeared that the Pi was connected to the wireless network and controllable via TeamViewer, but would fail to deliver information to the server architecture. We speculated this may be the result of spotty coverage, as an Ethernet connection in the lab appeared to function perfectly fine. We were unable to replicate the Ethernet connection in the wild due to network constraints.

For testing device pickup across the nodes we placed the Raspberry Pis together on one desk in the lab and ran BlueHydra on each while observing the list of identified devices. This allowed for real time comparison of how each unique setup harvested information.

## Results

We observed that two identically outfitted nodes do not always pick up the same list of devices. This creates a certain degree of uncertainty in our results. This is possibly due to the manner in which Ubertooth scans for Bluetooth, but we were unable to pinpoint a solution.

In the process of building out a backend, we ran into various issues in the way Elasticsearch processes redundant data points. We had to change our script to make sure each entry sent to the Elasticsearch server was unique. There are a few ways that the Elasticsearch Python library can be used to send data to our instance. We ended up using the *create* function, which is just a slight variation of the *index* function. However, making sure the ID's included in the Elasticsearch statement were unique was up to our script. Uniqueness was created by using the *time.time()* function paired with the hostname of the device. This makes sure each entry will not have future conflicts when a new node is added. We are currently able to track Bluetooth devices in both discovery and non-discovery mode as predicted in the beginning of this project.

## Conclusion

This project serves as proof-of-concept that tracking Bluetooth with nodes can be achieved. The most important caveat is the inconsistency of Blue Hydra picking up signals of Bluetooth devices. Each Raspberry Pi is configured identically using the same script and hardware dongles, but do not generate identically listings of devices. This poses a serious issue for the most basic premise of this project, and accuracy could be improved with better knowledge of how Bluetooth devices send packets. It is entirely possible a target device could cross multiple nodes and only be picked up by one.  In addressing the original research question of scalability, this inconsistency becomes concerning. We only examined the effects across three devices, but in adding more it is safe to assume that more issues would arise. We understand the Ubertooth and Bluetooth adapters are not perfect and cannot sniff everything, but our end goal is significant.

The second compounding issue is the way in which Elasticsearch handles the data input. Instead of creating individual entries for each time the device identifier is seen, the database “updates” the original entry with a new node location and time. This obviously makes the “tracking” portion difficult, as the data points that would be used to track a device through time and space are being constantly overwritten. Luckily, we were able to get around this issue by setting the ID sent with each Elasticsearch query is unique, as explained above. However, this was not the intended use. While Elasticsearch may work great in the aspect of scalability, the design of this system is not properly utilized for viewing historical records.

As the project stands currently, our set-up is capable of recording the last known time and location of a given Bluetooth identifier. The framework in place is promising but we feel there may be better luck with a different database system. The use cases for Elasticsearch are different than our uses, but are very useful for their easy setup process and visualization tools through Kibana. We have achieved our ultimate goal with this project of tracking both discoverable and non-discoverable devices, with the ability to visualize where they have been. This project can certainly be used in a real-world scenario, with a shift to a cloud hosting provider. We look forward to seeing where this project can go in the near future.

## Further Work

During our testing, we discovered that Bluetooth devices are not consistently identified by the Ubertooth. It is unclear if this is due to an issue with the software we are using (BlueHydra) or the hardware (the Ubertooth). More testing on this would be appropriate. In cases of Bluetooth Low Energy and newer protocols, the utility of an Ubertooth to detect non-discoverable devices may not be as strong as it had been in previous years. Through research on the differences between iOS and Android Bluetooth usage, it may become clearer as to why we obtained the results that we ultimately did. It is suggested that future groups conduct a more thorough side by side analysis to discover the extent of this issue.

We also came across many issues with the reliability of wireless connectivity given our network situation, which was sometimes stretched through a few floors of our building. The script can be optimized better to include safeguards for network-connectivity or database parsing issues. This project should be moved to a cloud-based host so there would be more reliable connections instead of relying a network that is physically far away. This was one of our original intentions, but due to time constraints fell short. Moving it up to a dedicated host will allow more nodes to be added, and expansion of the project overall can be continued. This project serves as a great start to a Bluetooth-device tracking framework, and we hope more accurate and lower-cost nodes can be created for expansion of the project in the future.

## Appendix

### Appendix A (setup.sh)

This script is designed to install all the dependencies for the scanner script to run. It is a shell script and should be run by first running “sudo chmod +x FILENAME.sh” followed by “sudo ./FILENAME.sh”.

```
#!/bin/bash
if [ "$EUID" -ne 0 ]
  then echo "Please run as root"
  exit
fi
cd ~
mkdir installations
mkdir downloads
sudo apt-get upgrade
sudo apt-get update
sudo apt-get install git nano cmake make libusb-1.0-0-dev gcc g++ libbluetooth-dev
pkg-config libpcap-dev python-numpy python-pip python-pyside python-qt4 bluez bluez-
test-scripts python-bluez python-dbus libsqlite3-dev sqlite3 ruby-dev bundler vim
sudo pip install --upgrade pip
pip install elasticsearch
wget https://github.com/greatscottgadgets/libbtbb/archive/2017-03-R2.tar.gz -O
libbtbb-2017-03-R2.tar.gz
tar xf libbtbb-2017-03-R2.tar.gz
sudo rm -rf libbtbb-2017-03-R2.tar.gz
cd libbtbb-2017-03-R2
mkdir build
cd build
cmake ..
make
sudo make install
sudo ldconfig
rm -rf libbtbb-2017-03-R2
cd ~
git clone https://github.com/greatscottgadgets/ubertooth.git
cd ubertooth/host
mkdir build
cd build
cmake ..
make
sudo make install
sudo ldconfig
rm -rf ~/ubertooth
cd ~/installations
git clone https://github.com/pwnieexpress/blue_hydra.git
cd blue_hydra
bundle install
cd bin
echo Please enter hostname!
read newhost
echo $newhost > /etc/hostname
echo $(hostname -I | cut -d\  -f1) $(hostname) | sudo tee -a /etc/hosts
echo The device will now reboot.
sudo shutdown -r now
```

Syntax highlighting courtesy of hilite.me.

## Appendix B (spypies.py)

This script is designed to continually run and parse the data from blue hydra. It is a python program written in python version 2.7 and should be run using the command “python FILENAME.py”.

```
from datetime import datetime
from elasticsearch import Elasticsearch
from time import sleep
import sqlite3
import os
import shutil
import socket
from time import gmtime, strftime
import time

es = Elasticsearch(['192.168.10.34:9200'])

#Always place script in following directory: /home/pi/installations/blue_hydra/bin/

#Constant variable locations DO NOT CHANGE
class paths:
    BH = '/home/pi/installations/blue_hydra/bin/blue_hydra'
    DB = '/home/pi/installations/blue_hydra/bin/blue_hydra.db'
    COPY_DB = '/home/pi/installations/blue_hydra/bin/blue_hydra_COPY.db'

#Creates dictionary list of names for what columns are in DB and creates ability to be
paired with a value
def dict_factory(cursor, row):
    d = {}
    for idx, col in enumerate(cursor.description):
        d[col[0]] = row[idx]
    return d

#Parses data from BlueHydra DB & outputs JSON file with names & values
#Maybe add a while true statement to continue and then break when results is populated
def blueHydraParser():
    start = datetime.now()

    #Checks if main database is empty
    connection = sqlite3.connect(paths.DB)
    connection.row_factory = dict_factory
    cursor = connection.cursor()

    cursor.execute("select * from blue_hydra_devices")
    results = cursor.fetchall()
    while not results:
        print "Database is empty, running Blue Hydra for another minute"
        os.system("sudo timeout -s 2 60s " + paths.BH + "  -d")

    else:
        #Duplicates file so we are not operating on live database
      shutil.copyfile(paths.DB, paths.COPY_DB)
```

```
          print("Database File Copied")
        os.remove(paths.DB)
        print("Original Database Deleted")
        print(os.path.isfile(paths.DB))

          connection = sqlite3.connect(paths.COPY_DB)
          connection.row_factory = dict_factory
          cursor = connection.cursor()

          cursor.execute("select * from blue_hydra_devices")

          #Fetches all rows from table
        results = cursor.fetchall()
          #Gets the hostname of the device
        hostname = socket.gethostname().lower()

          current_time = time.time()

          # This sends the data to Elasticsearch
          # MAKE A TRY STATEMENT TO FIX DISCONNECT ISSUE
          for bt_capture in results:
              bt_capture.update({'hostname':hostname})
              es.create(index="spypi", doc_type='packet', id=hostname+str(time.time()),
body=bt_capture)
              sleep(.1)
          os.system("sudo rm " +  paths.COPY_DB)
          print("Copied DB Deleted")
          connection.close()

def main():
    print("Starting Blue_Hydra for 1 min\n")
    os.system("sudo timeout -s 2 60s " + paths.BH + "  -d")
    print("Blue Hydra has ran\n")
    print("Starting parse function\n")
    blueHydraParser()
    print("Parse function ended @ " + strftime("%Y-%m-%d %H:%M:%S", gmtime()) + "\n")

    #time_elapsed = datetime.now() - start
    #print("Time {}".format(time_elapsed))

while True:
    main()
```

Syntax highlighting courtesy of hilite.me.

## References

Adafruit's Raspberry Pi Lesson 3. Network Setup. (n.d.). Retrieved April 02, 2018, from https://learn.adafruit.com/adafruits-raspberry-pi-lesson-3-network-setup/setting-up-wifi-with-occidentalis

"Elasticsearch." *Open Source Search & Analytics · Elasticsearch*, www.elastic.co/products/elasticsearch.

"Encyclopedia." *2.4 GHz Band Definition from PC Magazine Encyclopedia*, www.pcmag.com/encyclopedia/term/58396/2-4-ghz-band.

Express, Pwnie. "Pwnieexpress/blue_hydra." *GitHub*, 1 Mar. 2018, github.com/pwnieexpress/blue_hydra.

GeoJSON. "The GeoJSON Specification." *GeoJSON*, geojson.org/.

"Greatscottgadgets/Ubertooth." *GitHub*, 27 Dec. 2017, github.com/greatscottgadgets/ubertooth.

Hirose, Mariko. "Newly Obtained Records Reveal Extensive Monitoring of E-ZPass Tags Throughout New York." American Civil Liberties Union, American Civil Liberties Union, 29 May 2015, www.aclu.org/blog/privacy-technology/location-tracking/newly-obtained-records-reveal-extensive-monitoring-e-zpass.

How it works | Bluetooth Technology Website. (n.d.). Retrieved October 23, 2017, from https://www.bluetooth.com/what-is-bluetooth-technology/how-it-works

Jimb0. "Bluetooth Basics." Bluetooth Basics, Sparkfun Electronics, learn.sparkfun.com/tutorials/bluetooth-basics/how-bluetooth-works.

mxs. "Bluetooth: Defining NAP + UAP + LAP." *Nccgroup.trust*, 13 Feb. 2012, www.nccgroup.trust/us/about-us/newsroom-and-events/blog/2012/february/bluetooth-defining-nap-uap-lap/.

Walkom, M. (2018, February 16). Custom Region Maps in Kibana 6.0. Retrieved March 23, 2018, from https://www.elastic.co/blog/custom-region-maps-in-kibana-6-0

"Welcome to Python.org." *Python.org*, Python Software Foundation, www.python.org/.