Wearable Device Security: Hacking Wearables

Nate Wood

University of Nebraska at Omaha

Table of Contents

[Wearable Device Security: Hacking Wearables 3](#_Toc480827147)

[Introduction 3](#_Toc480827148)

[Bluetooth Low Energy (BLE) 3](#_Toc480827149)

[Pairing and encryption with BLE 3](#_Toc480827150)

[BLE Frequency Hopping 4](#_Toc480827151)

[Bypassing the encryption of BLE. 6](#_Toc480827152)

[Ubertooth technical specifications. 7](#_Toc480827153)

[Mitigation and Prevention 7](#_Toc480827154)

[Application of research. 7](#_Toc480827155)

[Using Ubertooth 8](#_Toc480827156)

[Packet Capture with Ubertooth 8](#_Toc480827157)

[Identifying a successful connection 9](#_Toc480827158)

[Injecting packets into the device 9](#_Toc480827159)

[Conclusion 11](#_Toc480827160)

[References 12](#_Toc480827161)

[Tables 14](#_Toc480827162)

[Figures 16](#_Toc480827163)

# Wearable Device Security: Hacking Wearables

## Introduction

This research paper aims to address security concerns with wearable devices and their communication. The Bluetooth Low Energy protocol is used as a communications channel between the wearable and the paired device (e.g. phone). I will be using an Ubertooth One to intercept communications between the wearable and its paired device. The Ubertooth One is an open source Bluetooth packet capture device. Its code is freely available on GitHub (Great Scott Gadgets, 2017). The wearable device being used is a MetaWear CPRO. The CPRO has open API and data sheets which is helpful for the research. After observing the connection information, I will attempt to use packet injection to send false information to the wearable or the paired device. My secondary objective is writing a packet capture command which auto detects the successful connection request and records its data into another file.

Much of my research revolves around an existing research topic from Mike Ryan’s white paper on Bluetooth Low Energy: *Bluetooth: With Low Energy Comes Low Security.* My research on BLE security aims to reflect on the research covered in Ryan’s whitepaper, and expand on the application of the research by adding an attack vector.

## Bluetooth Low Energy (BLE)

Many wearable devices operate on the Bluetooth low energy protocol ( Bluetooth SIG, 2017). Bluetooth low energy (BLE) is a power efficient protocol which powers wearables, Internet of Things devices, heart rate monitors, smart watches, TVs, thermostats and many other devices. BLE key features include:

* Industry-standard wireless protocol that allows for interoperability across platforms
* Ultra-low peak, average and idle mode power consumption
* Standardized application development architecture eases development and deployment time and cost
* Allows for government-grade security with 128-bit AES data encryption ( Bluetooth SIG, 2017)

### Pairing and encryption with BLE

BLE runs on three different pairing methods: Just Works, Passkey Entry, and Out of Band technology (NIST, 2012). Each pairing begins with key transport instead of a key agreement like the Bluetooth protocol uses. The key generation is performed by the host device. Since it uses key transport, a key distribution step is required during the BLE pairing.

As shown in Figure 1 Bluetooth Low Energy Pairing (NIST, 2012), LE pairing begins with the two devices agreeing on a Temporary Key (TK), whose value depends on the pairing method being used. The devices then exchange random values and generate a Short Term Key (STK) based on these values and the TK. The link is then encrypted using the STK, which allows secure key distribution of the Long-Term Key (LTK), Identity Resolving Key (IRK), and Connection Signature Resolving Key (CSRK) (NIST, 2012).

Generating the BLE key and distribution is a fairly simple process. After the STK has been exchanged, and the communications channel has been encrypted with the STK, the master and slave devices can generate and exchange the LTK, CSRK, and IRK. Two options are available for key generation. The first has the device generate two random 128-bit values and store them in a local database, or “database lookup” in the specification. The second option uses a single 128-bit random static value called the Encryption Root (ER) along with a 16-bit Diversifier (DIV) which is unique to each trusted device to generate keys, or “Key Hierarchy” in the specification. The formulas for the key generation processes for the LTK, CSRK, and IRK can be found in Figure 7 Key Generation Formulas (NIST, 2012).

Just Works pairing for BLE is the least secure of the three – the TK is set to all zeros (0x00). This means the eavesdropper does not need to guess the TK to generate the STK. Since the TK is known to the potential attacker – there is no Man in the Middle (MITM) protection for Just Works.

Passkey Entry is the second most secure way to pair BLE devices. This method requires that at least one device supports a keyboard input and the other device a display output. It uses a six-digit code for a TK. This is how my smartwatch (Motorola 360) pairs with my smartphone. The Moto 360 generates a 6-digit key and I confirm the key on my smartphone. Per NIST, the 6-digit passkey provides a maximum of 20 bits of entropy (2012). This process provides an authenticated LTK.

Out of Band (OOB) pairing requires that both devices have similar NFC or tethering technology. In this process the TK is passed via the shared OOB protocol. This method of pairing also provides an authenticated LTK. The pairing protection against MITM attacks is based upon the OOB technology itself.

Encryption in BLE uses AES-CCM cryptography – it generates a 128-bit key from a 128-bit plaintext data using the AES-128-bit block cipher. Even though BLE uses 128-bit AES encryption the pairing process leaves it open to MITM attacks.

### BLE Frequency Hopping

To increase privacy, BLE has incorporated a feature that changes the Bluetooth device channel on a frequent basis. This feature is commonly known as frequency hopping.

BLE operates in the 2.4 GHz Industrial Scientific Medical (ISM) band and defines 40 Radio Frequency (RF) channels with 2 MHz channel spacing. There are two types of BLE RF channels: advertising channels and data channels. Advertising channels are used for device discovery, connection establishment and broadcast transmission, whereas data channels are used for bidirectional communication between connected devices. (Gomez, Oller, & Paradells, 2012).

There are three advertising channels and thirty-seven data channels. The three advertising channels are 37, 38, and 39 (Figure 2 BLE Channels) (Blackberry, 2013). When a BLE device is turned on or is looking for a connection it starts to broadcast on one of the three advertising channels. This information is used for intercepting the communications between the wearable and host device. The Ubertooth listens to all the data coming in on one of the advertising channels and if it sees a connection request it scrapes the data from the packet and starts to follow the connection. The Ubertooth software follows the BLE device after it hops channels to another frequency by observing the information inside the connection packet. A BLE packet format contains an 8-bit Preamble, a 32-bit Access Address, a 2 to 39-bit Protocol Data Unit (PDU), and a 24-bit Cyclic Redundancy Check (CRC) (see Figure 6 Bluetooth Low Energy packet format (Ryan, USENIX WOOT, 2013)).

Frequency hopping aims to prevent eavesdropping on the BLE connection. Each connection is unique in the way it accomplishes frequency hopping. Several variables must be established for both the host and device to be able to frequency hop at the same interval and onto the same channels. These variables include:

1. Hop interval
2. Hop increment
3. Access address
4. CRC (Cyclic Redundancy Check) initializer

To follow a connection, we need the four above parameters. The BLE channel hopping sequence is fairly simple protocol. After we determine the hop interval and increment we can successfully follow a connection when it hops to another frequency. This formula is *nextChannel = channel + hopIncrement (mod 37)* (see Figure 5 Next Channel Formula (Ryan, USENIX WOOT, 2013)).

The hop interval is also known as a dwell time – this is the amount of time the device and host operate on a single 2.4Ghz channel before hopping to the next predetermined channel. According to Mike Ryan you can recover the hop interval by observing that the hop sequence completes a full cycle once every 37 x 1.25 x *hopInterval* milliseconds (see Figure 3 Hop Interval Formula) (Ryan, USENIX WOOT, 2013).

The hop increment is determined by observing two channels and waiting to see how long it takes for the packets to start arriving at the second channel. Listen on channel 0 and when a packet is observed switch to channel 1 and measure the amount of time to see another packet. The formula to determine the hop increment is *channelsHopped* equals the time delta divided by 37 x 1.25 ms (see Figure 4 Hop Increment Formula (Ryan, USENIX WOOT, 2013)).

To determine the device access address, you can observe an arbitrary data channel to listen for empty data packets. Since the BLE specification requires the device and host transmit a packet on each channel hop to, you can sniff the access address passively. Each data packet contains a 16-bit header, 0-37 octets of payload (PDU), and a 24-bit CRC. By reading the data stream and observing an empty packet, we can then use the prior 32-bits as a candidate access address for the device. After observing a set amount of these access address candidates, we can use the candidate which has a predetermined number of matches as our access address and follow it.

Finally, to recover the CRC (Cyclic Redundancy Check) initializer which is also known as the CRCInit, you must observe a set number of candidate packets. This process is similar to the access address candidate process. Each connection has a unique CRCInit. When a new 24-bit CRC is observed from a packet capture, it is used to seed linear feedback shift register (LFSR) in reverse order to determine the candidate CRCInit.

### **Bypassing the encryption of BLE.**

Bluetooth Low Energy uses a proprietary key exchange, as covered earlier in this paper. This leaves the technology prone to weaknesses against passive eavesdroppers. The encryption which BLE uses, AES-CCM, is known to be secure. The attacks used today are against the key exchange rather than the encryption protocol. The master and slave devices must establish a LTK to be used for encrypted communication sessions. To establish the LTK the two devices must first establish a TK, whose value varies based upon the pairing mode chosen. The values are exchanged in plaintext, except for the TK. The fact that confirm value, Channel interval and hop increment can be determined allows us to derive the TK from the plaintext values. The three pairing modes, Just Works, Passkey Entry, and Out of Band have different methods of defining the TK values.

* Just Works: The TK is always 0.
* Passkey Entry: The TK is a value between 000000 and 999999. This value is padded to 128-bits
* OOB: a 128-bit value is exchanged OOB

(Ryan, USENIX WOOT, 2013)

Obviously, it is not an issue to guess the Just Works TK considering it is always 0. The 6-digit Passkey entry TK can be brute forced, considering there are a finite number of possible combinations. According to Mike Ryan, a TK can be cracked in less than 1 second on a single core of an Intel i7 CPU (Ryan, USENIX WOOT, 2013). It’s so easy it is as if you don’t even have a TK in place. After the TK is confirmed we can use it to decrypt the STK. The STK is used to establish a link-layer encrypted session, at which point we can decrypt the session and recover the LTK. After the key exchange process is completed the master and slave devices communicate using the LTK for encryption. If we can recover the LTK as described before, we will be able to decrypt the future communications made between the two devices. Since the initial variable for this security protocol is so weak, it leads to the entire compromise of the key exchange process and rendering the encryption protocol useless. At this point it doesn’t matter what encryption protocol the device chooses to use. It is noteworthy to consider that the OOB pairing methodology will not be able to be compromised in this same fashion. A well-chosen OOB key will render brute forcing useless. It has been my experience that most devices use the Just Works or the Passkey pairing methods. Those two pairing methods are easier to implement because they do not require any special hardware to exchange the OOB key. These attacks are even easier to accomplish if you consider that you can accomplish the brute forcing offline if the attacker records the key exchange. Any future conversations which use the LTK, which was decrypted, can be recorded and decrypted offline if the initial encryption setup was recorded.

## **Ubertooth technical specifications.**

How does an Ubertooth capture the BLE packets? The Ubertooth is a USB dongle with a RF frontend. The radio chip on the Ubertooth was reconfigured to demodulate BLE parameters. The Ubertooth has been programmed to make the decisions on whether to jump to additional channels. It is tasked with receiving the data, de-whiten it, parsing the header, validating the CRC, and logging the information gathered to the host PC. The reason all the data is processed on the Ubertooth itself instead of being processed on the host PC is latency. Timing is everything and the latency of passing the data through USB to the PC to be processed might take too long and the connection could be lost. The Ubertooth’s modem is configured “to demodulate GFSK with a frequency offset of 250 kHz and a data rate of 1 Mbit” (Ryan, USENIX WOOT, 2013). The start of the transmission is identified by a known 32-bit access address. It then sends the captured bits to the PC to be logged into a PCAP. This PCAP data is then imported into Wireshark to be analyzed.

## Mitigation and Prevention

If a connection is already established, then the master and slave Bluetooth devices do not need to reestablish a key using the key exchange protocol. This would prevent the attacker from passively sniffing the connection request packet information and key exchange process. Since a session-specific nonce is used and exchanged at the beginning of each session, it is impossible to decrypt the session even if the LTK is known.

To counter the mitigations, you could potentially send a reject connection message on the link layer. This would force the key negotiation to happen again and you could compromise the connection as described earlier in the paper. Alternatively, you could also send a packet flood or jam the connection, which would force the master and slave devices to reconnect and re-establish secure connections. In this case, we could sniff the nonce which allows us to decrypt the session. Per Mike Ryan, they have a jammer which “follows along the channel hopping sequence and injects random noise (output from an LFSR). In practice this kills connections almost instantly” (Ryan, USENIX WOOT, 2013). If you were able to make the session restart or re-establish its key exchange process, you would be able to sniff the data.

## Application of research.

In the following portion of the research paper I will discuss how I conducted my case studies and the application of the research. Using Ubertooth is a straightforward process if you have an assembled board provided to you. They also have a guide to assemble your own Ubertooth, however it is a very involved process – you would have to order a PCB to be made, stencil for the soldering paste, and the antenna and other parts as well (Great Scott Gadgets, 2017). Other hurdles for the application of the research include BLE packet injection, identifying a successful connection, and interpreting the contents of packet captures (PCAP).

### Using Ubertooth

Ubertooth is available in either a kit or a preassembled board. I was given a preassembled Ubertooth. It already had the programming flashed onto it. I was also given an Ubuntu VM for Ubertooth testing. This was configured with Wireshark and the Ubertooth software. I will assume that anyone configuring their own test lab will have the same configuration provided.

Ubertooth has five major modes. (see Figure 8 Ubertooth-btle help page)

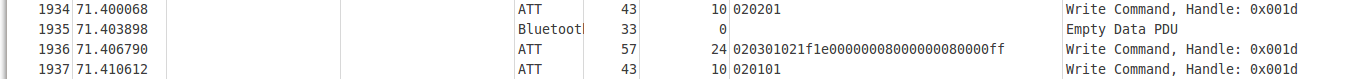
* -f follow connections
* -p promiscuous: sniff active connections
* -a[address] get/set access address
* -s<address> faux slave mode, using MAC address
* -t<address> set connection following target

These determine how the chipset will work. For this exercise, I will be using the “sudo Ubertooth-btle -f -c /tmp/p\*” command to follow connections. The default channel to listen on is 37, which is one of the advertising channels for BLE. When I try to connect the master device to the slave device the Ubertooth will be listening on channel 37. If the connection happens on channel 37 then the Ubertooth will start following the connection. When I run the command the -c will output the Packet Capture (PCAP) files to the /tmp/p\* file. I will use this file to find packets which correspond to the device. For this exercise, I will be using the MetaWear application on my Android phone to send a command to the CPRO device. For testing I will be commanding the red LED to turn on so I specifically know what packet to look for when I attempt to fuzz the BLE connection and turn on the LED from the Ubuntu VM.

### Packet Capture with Ubertooth

When I started the packet capture using the above parameters, I saw many different packets fly by in the terminal. This was too fast for humans to read. When I tried to connect the CPRO slave device I did not see a change in the frequency of packets being captured. I mostly saw the packets that were running by before. I assumed this was because the Ubertooth was not listening on the correct channel. Since the default listening channel is on 37, the advertising channel used could have been on 38 or 39. Trying a few more times produced results. I could see packets flying by even faster than before – this is how I knew that the packet capture had worked. To validate the PCAP was working, I sent a set of commands from my phone to the CPRO device – turn red LED on, turn LED off – twice.

After I had a successful packet capture I examined the PCAP results in Wireshark. The PCAP results reflected a large amount of noise, even capturing in my house which is an isolated area unlike UNO or another public place. Even with the extra noise, I could still distinctly see the commands I sent from my phone. You can see the results in the screen shots below (also see Figure 9 Turn on red LED PCAP, and Figure 10 Turn off LED PCAP) \*\*I have not been able to successfully recreate the below packet sequence. I will need to do further research on the packet format to determine what to replay in a packet injection.





I could see those 4 packets twice in the packet capture in the same sequence. This was the evidence that I had successfully captured the commands I sent. The payload of the packets I captured was still not clear to me. My hypothesis was that the packet labeled 1936 was the actual command which sent the red color command to the LED because it had more data in the “value” column of the PCAP.

The next test I will be running is trying to see if I can get a successful connection, then send a new sequence of commands. Turn red LED on, turn LED off, turn green LED on, turn LED off, turn on red LED, turn off LED, turn on blue LED, turn off LED, turn on red LED, turn off LED. I hope to figure out exactly the payload of the packet for turning on a red LED. By running this command sequence, I may also be able to determine what the packet values are for turn on blue/green LED as well as turn of LED.

### Identifying a successful connection

While doing my research, I found that it can be somewhat difficult to understand when a successful connection has been established. Opening the PCAP while it was actively listening would kill the connection, so using Wireshark to view the PCAP while the connection streamed data was out of the question.

I set out to figure out a way to check to see if a connection was established without interrupting the stream. My initial idea was to write some code which would run in the background of the Ubertooth packet capture, potentially as a command line option when launching the program. While reading the Ubertooth code, it appeared that this might be a complex and difficult option. Exploring other options led me to discover that I could copy the file while it was streaming data, like a snapshot of data which had been written at the time, without interrupting the connection.

I used python version 2.7.3 to write the program. After trying to read the copy of the PCAP file I realized it was not in plaintext. To make the PCAP readable it needed to be converted to plaintext. I used tshark to convert the PCAP into a .txt file. After converting the file I used grep to find a specific string in the PCAP.txt file: “CONNECT\_REQ”. This connect request indicated that a successful connection had been established. If the program finds the string in the provided file it will print the line plus relevant lines before and after the string to show the complete packet on the screen. It also tees the output to a <input>\_results.txt for future reference. Please see Table 1 Connection\_tester.py to review code.

### Injecting packets into the device

Throughout my research I have been thinking about how I could inject packets back into the data stream while the connection was established. According to Mike Ryan, BLE packet injection has been proven as a proof of concept (POC). Ryan’s white paper indicated that they successfully used the CC2400 chipset on the Ubertooth to send advertisement messages which broadcast the existence of the device. These advertisement messages were received by a PC running the Linux Bluetooth stack “bluez” during a scan for Bluetooth devices.

Ryan theorizes that the injection operation is similar to receiving packets, but in the opposite direction. They configured the Ubertooth to operate in a buffered mode, which is slower than its unbuffered mode, but is less stable. When they conducted the POC they crafted the packet on the LPC (the microcontroller used by Ubertooth) they set the AdvA (advertising address) to a user defined MAC address, and the packet CRC is calculated (Ryan, BLE Fun with Ubertooth Sniffing, 2014). They also indicate that for a more targeted and sophisticated attack will need to use the unbuffered mode to accommodate tight timing require for packet injection. The white paper did not indicate how to accomplish packet injection, just that it could be possible.

While doing additional research on the subject, I decided to try to ask the Ubertooth community which has a IRC on freenode. When I logged onto freenode I asked if anyone had used Ubertooth to inject packets into a device. Much to my surprise Mike Ryan responded to my question! The response I received, however, was a bit disappointing – he indicated that “it's physically possible to do this with ubertooth, at least for BLE, but the software and firmware do not support it” (see Table 2 Freenode Chat with Mike Ryan). Ryan added that he has spent considerable time on the subject, but has not got anything working in any stable fashion.

#### Additional packet injection theory

I theorize that packet injection could be accomplished using the connection information sniffed using the Ubertooth from the connection made between the BLE master and slave devices. The transmission of the packet could be through a secondary Bluetooth device – like the built-in Bluetooth in my laptop or a USB Bluetooth dongle. I would use Ubertooth to capture the packet sequence I believe will turn on the red LED, then replay the packets using a packet replay tool. This tool could be BLE-Replay as it supports Bluetooth Low Energy packet replay.

Replaying the packet would only be half the equation. The BLE target device will also need to be on the correct channel for the device to properly receive the packets being injected. I believe I could pick a data channel, for example channel 5, and replay the packet sequence in a loop. My theory is the BLE device would eventually hop to the correct channel and then receive the packets being replayed.

I found the BLESuite, which is a python package that provides an easy way to test BLE devices. This is an open source tool which is available on GitHub (NCCGroup, 2017) that is designed to work with HCI logs from an Android device. I have attempted to install the BLESuite but have had compatibility issues with my installation on the Ubuntu VM that I have been using for my research. As such, I have not yet been able to test the functionality of the tool.

## Conclusion

BLE is a widely-used protocol which does not have very much thought put into security. The encryption protocols used are not the root cause of the security flaws, rather the three pairing protocols are.

The least secure of the three – Just Works – puts no effort into protection against passive sniffing connections. If just the initial connection request is observed, you can follow and decrypt the entire BLE connection contents. The effort to snoop those connections is trivial. My recommendation for devices manufacturers who are concerned with security of their BLE devices using Just Works pairing is to update the pairing. I believe that OOB pairing would be much more secure since you would have to be within range of the OOB pairing and be able to intercept the pairing to sniff the connection information. The OOB connection STK could be a stronger key than the 0s that Just Works uses. If this change was made, then we could rely on the encryption protocol instead of the pairing method.

The Passkey pairing method could also be more secure. I would increase the length of the passkey, or possibly keep the 6-character length and allow alphanumeric characters to be used instead. The passkey in this method is the weakest link in the privacy chain. It can be cracked using open source tools like Crackle (available from GitHub (Ryan, Crackle, 2016)) easily – if you can decrypt the STK passkey then you can use that to generate the LTK as well. Increasing the strength of the STK will increase the security of the pairing method. This improved Passkey pairing method would be better but OOB would still be preferential.

When I started this project, I set out to use Ubertooth to do packet injection. I was not able to complete that objective due to the fact the software and firmware provided with the Ubertooth was not capable of the task. I have however created a better way to indicate whether a connection was successfully observed with the Ubertooth. This method would be difficult to adapt to the codebase of the Ubertooth at this time since it relies on the file to be converted to a .txt file. If I could translate the process to be able to search the PCAP file then it would be possible that it could be adapted to run while the stream of data is in place, and then alert the user that a connection has been established.

References

Bluetooth SIG. (2017). *Bluetooth Low Energy*. Retrieved from Bluetooth: https://www.bluetooth.com/what-is-bluetooth-technology/how-it-works/low-energy

Blackberry. (2013, 04 05). Bluetooth LE Channels. *BlackBerry 10 - Bluetooth LE primer for developers*.

Bluetoth SIG. (2017). Security, Bluetooth Low Energy.

Gomez, C., Oller, J., & Paradells, J. (2012, 12 09). *Overview and Evaluation of Bluetooth Low Energy: An Emerging Low-Power Wireless Technology.* Retrieved from MDPI: http://www.mdpi.com/1424-8220/12/9/11734/htm

Great Scott Gadgets. (2017). *Assembling-Hardware*. Retrieved from www.github.com: https://github.com/greatscottgadgets/ubertooth/wiki/Assembling-Hardware

Great Scott Gadgets. (2017, March 09). *greatscottgadgets/ubertooth*. Retrieved from GitHub: https://github.com/greatscottgadgets/ubertooth

NCCGroup. (2017). *BLESuite Github*. Retrieved from https://github.com: https://github.com/nccgroup/BLESuite

NCCGroup. (2017). *Introducing BLESuite and BLE-Replay python tools*. Retrieved from www.nccgroup.trust: https://www.nccgroup.trust/us/about-us/newsroom-and-events/blog/2016/september/introducing-blesuite-and-ble-replay-python-tools-for-rapid-assessment-of-bluetooth-low-energy-peripherals/

NIST. (2012, 06 12). Guide to Bluetooth Security.

Ryan, M. (2013, September 26). *USENIX WOOT*. Retrieved from Lacklustre.net: https://lacklustre.net/bluetooth/Ryan\_Bluetooth\_Low\_Energy\_USENIX\_WOOT.pdf

Ryan, M. (2014, January 27). *BLE Fun with Ubertooth Sniffing*. Retrieved from blog.ice9.us: http://blog.ice9.us/2014/01/ble-fun-with-ubertooth-sniffing.html

Ryan, M. (2016). *Crackle*. Retrieved from GitHub: https://github.com/mikeryan/crackle/

Tables

Table Connection\_tester.py

|  |
| --- |
| #run this script after connecting the device to see if the connection request packet was observed. |
| #Pass the file name of the current packet capture into the script for a file conversion and to search the file. |
| # The PCAP must be run from /tmp not ~/tmp for the CONNECT\_REQ to be captured properly. |
| import sys |
| import os |
|  |
| #first arg of script is file name |
| f = open(sys.argv[1]) |
| file = sys.argv[1] |
| convert = 'tshark -V -r ' + file + '> ' + file + '.txt' |
| command = 'cat ' + file + '.txt | grep CONNECT\_REQ -A 13 -B 26 | tee ' + file + '\_results.txt' |
|  |
| #convert the pcap to .txt |
| print ('Converting ') + sys.argv[1] + (' to a .txt file...\n') |
| os.system(convert) |
|  |
| #search the converted pcap for CONNECT\_REQ |
| print ('Searching for CONNECT\_REQ in: ') + sys.argv[1] + ('.txt\n') |
|  |
| if 'CONNECT\_REQ' in open(file + '.txt').read(): |
| print 'Connect request found!\n' |
| os.system(command) |
| # os.system('cat test.txt | grep CONNECT\_REQ -A 13 -B 26') |
| sys.exit('End of file - exiting program') |
| else: |
| print 'Could not find Connect request\n' |

Table Freenode Chat with Mike Ryan

|  |
| --- |
| [19:22] == woody\_ [46bb138a@gateway/web/freenode/ip.70.187.19.138] has joined #ubertooth  [19:22] <woody\_> Hey there  [19:23] <woody\_> Has anyone used ubertooth to inject packets to a device?  [19:23] <mikeryan> it's physically possible to do this with ubertooth, at least for BLE, but the software and firmware do not support it  [19:25] <woody\_> Hey mike ! I am a grad student and am researching this subject - I've read your white papers on the subject  [19:25] <woody\_> thanks for responding :)  [19:25] <woody\_> so injection with a stock ubertooth is not possible ?  [19:27] <mikeryan> correct, our released firmware does not support it  [19:27] <woody\_> Could I possibly write a packet and send it over BLE with the ubertooth with another piece of software?  [19:28] <mikeryan> the stock firmware can send BLE advertising packets  [19:28] <mikeryan> it can't do BLE link layer connections  [19:30] <woody\_> oh - well that is somewhat disappointing - have you done more exploration on the injection since releasing the research?  [19:32] <mikeryan> i have spent considerable time on it, but i haven't got anything working in any stable fashion  [19:32] <mikeryan> it's always been a side project, never got any official funding or time to work on it  [19:35] <woody\_> Do you have a repo or files you would be willing to share for use in my research project? |

Figures

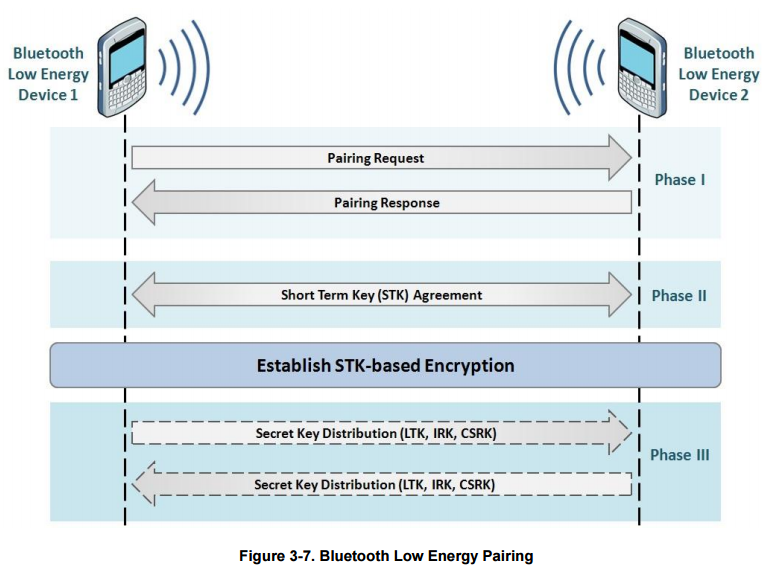


Figure Bluetooth Low Energy Pairing (NIST, 2012)

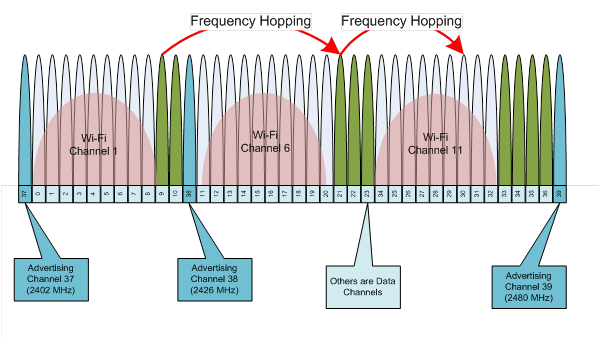


Figure BLE Channels (Blackberry, 2013)



Figure Hop Interval Formula (Ryan, USENIX WOOT, 2013)

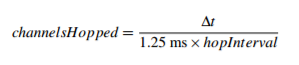


Figure Hop Increment Formula (Ryan, USENIX WOOT, 2013)



Figure Next Channel Formula (Ryan, USENIX WOOT, 2013)

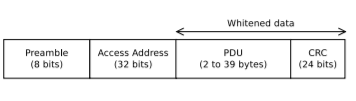


Figure Bluetooth Low Energy packet format (Ryan, USENIX WOOT, 2013)

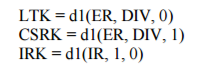


Figure Key Generation Formulas (NIST, 2012)

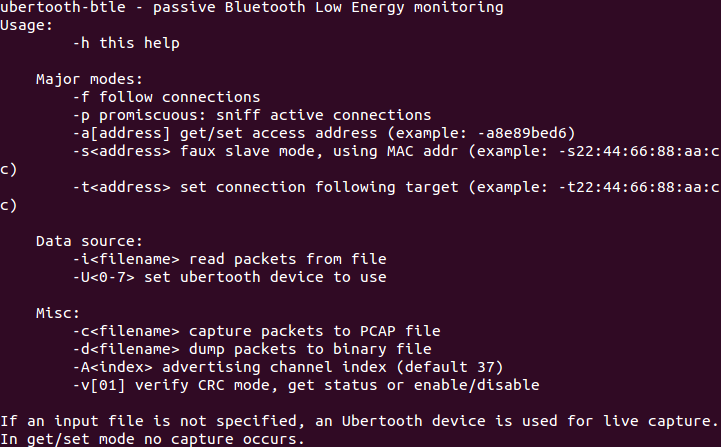


Figure Ubertooth-btle help page

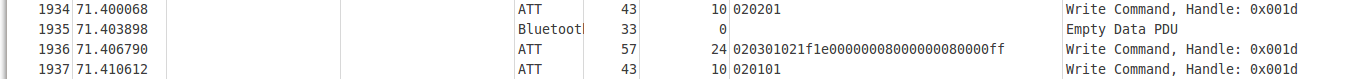


Figure Turn on red LED PCAP



Figure Turn off LED PCAP