Analyzing the Security of Bluetooth Low Energy

Seth Sevier
Network and Computer Security
SUNY Polytechnic Institute
Utica, New York 13502
Email: seviers@sunyit.edu

Ali Tekeoglu
Network and Computer Security
SUNY Polytechnic Institute
Utica, New York 13502
Email: ali.tekeoglu@sunyit.edu

***Abstract*—Internet of Things devices have spread to near ubiquity this decade. All around us now lies an invisible mesh of communication from devices embedded in seemingly everything. Inevitably some of that communication flying around our heads will contain data that must be protected or otherwise shielded from tampering. The responsibility to protect this sensitive information from malicious actors as it travels through the air then falls upon the standards used to communicate this data. Bluetooth Low Energy (BLE) is one of these standards, the aim of this paper is to put its security standards to test. By attempting to exploit its vulnerabilities we can see how secure this standard really is. In this paper, we present steps for analyzing the security of BLE devices using open-source hardware and software.**

***Index Terms*—Bluetooth, Security, IoT, Ubertooth**

I. Introduction

Bluetooth Low Energy, also known as Bluetooth Smart, is a wireless communication protocol created as a part of the Bluetooth 4.0 standard. As the name implies it is designed for usage on devices with a strict power budget. The aim of this research is to explore the potential vulnerabilities that exist within Bluetooth Low Energy by leveraging a handful of tools designed for this purpose. This paper breaks down into five core ways an attacker would try to break into a system that implements Bluetooth Low Energy. Firstly, understanding how the protocol works and preparing a system to attack it. Secondly, sniffing out traffic as it passes through the air. Thirdly, breaking the encryption used to protect the data. Fourthly, analyzing the now unencrypted data to find target information. Finally, attacking connections to cause damage to data or deny service to legitimate users.

II. Inner Workings of Bluetooth Low Energy

Before attempting to research how to break a protocol, it first becomes necessary to understand the various nuances of what goes on under the hood. Bluetooth Low Energy is designed to broadcast in a range between -30 dBm to 0 dBm [1]. The typical operating range for this protocol is two to five meters, however devices have been made able to transmit up to 30 meters [1]. In practice the ideal transmission rate of this standard is 5-10 KBps however in reality the developer should only be transmitting the minimum amount of data required anyways, as holding transmission windows open will quickly drain small batteries commonly used in Bluetooth enabled IoT devices [1].

One of the two modes that a Bluetooth Low Energy device operates in, to communicate to other devices, is *broadcasting mode* or *communication mode*. Broadcasting mode is a one-way communication typically used to send advertising packets to all devices within range. Broadcasting can also be used to transmit data to many devices at once but this application is fairly rare to encounter. For devices that rely on connection for data transmission the only two purposes this mode has is to send advertisement packets and initiate pairing events.

Advertisement packets contain information about the broadcaster and indicate that the device is available for pairing. These packets are useful for host enumeration but otherwise are of little concern to an attacker. Pairing events are what the attacker cares about. When pairing occurs a *Temporary Key* is used to generate a *Long Term Key* which is then used to encrypt the communication stream. Once this has been established communications can now travel both ways between devices.

The Temporary Key cannot be securely transmitted at any point in the communicaiton but somehow both devices must have that key. There are three ways to establish this. The first way is the most common - Just Works Mode. The key is either set to a series of zeroes or transmitted in cleartext. This way does not provide any protection from Man in the Middle attacks. Another method will have a six digit key displayed either as an etching on the device or as a digital display and the user must type the number into the other device. Finally, Out Of Band (OOB) mode refers to any key transmission that does not involve the Bluetooth radio or the Passkey Display mode [1].

III. Related Work

Internet of Things devices utilize several wireless technologies, while Bluetooth is one of the most common among them [2]. Bluetooth has been adopted for many different use cases such as; indoor proximity systems [3], microlocation [3], earthquake monitoring systems [4], vehicle communication with mobile phones [5]. Data transferred by beacons and BLE systems may contain private user data that need to be protected from intruders and from indiscriminate use [6].

There are 3 methods designed into Bluetooth protocol to provide security: *(i)* Use of pseudo-random frequency hoping, *(ii)* Restricted authentication and *(iii)* Encryption [7]. Generic Bluetooth protocol includes three security modes: *(a)* Non-Secure, *(b)* Service-levels security and *(c)* Link-level security. However, there are still security concerns that need to be addressed. In [8], authors list vulnerabilities that persist even after the security features have been introduced. Some of them include optional or weak encryption, non-secure default settings, weak PIN use, insecure unit keys, flawed integrity protections and predictable number generation. There are other generic threats that Bluetooth is also prone to such as eavesdropping, man-in-the-middle attacks, data corruption, and Denial of Service. Authors [7] propose solutions that address some of the security flaws, which include: User understanding of the technology, centralized Bluetooth pairing policy implementation, use of non-discoverable mode or on-demand access/pairing and mandatory encryption use.

IV. Sniffing with Ubertooth

Ubertooth One [9] is a hardware device designed to sniff Bluetooth Low Energy traffic. Unlike ordinary WiFi, Bluetooth and Bluetooth Low Energy utilize a transmission mode known as Frequency Hopping Spread Spectrum, or FHSS, to communicate across 36 channels ranging from 2404 MHz to 2478 MHz [1]. While it is trivial to make a WiFi card support promiscuous mode to listen for traffic not destined for its own hardware address, standard Bluetooth chips simply cannot do so as they can only pay attention to one channel at a time. Ubertooth One is a much more powerful card that will attempt to listen to traffic broadcasted across the entire frequency range allotted for Bluetooth Low Energy communications. Listening to the entire spectrum at once will allow for the capture of communications from all devices in range.

Figure 1. The Ubertooth One device is far more powerful than the regular Bluetooth chip.

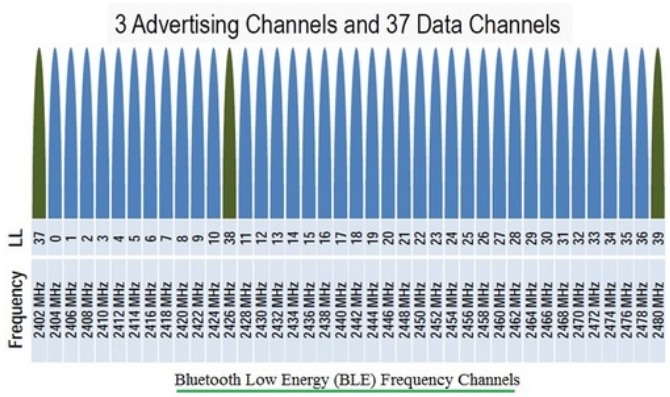

Figure 2. This is a graphical representation of the channels and frequencies used for BLE. Three channels (37, 38, 39) are reserved for broadcasting advertisement packets and 36 are reserved for FHSS communication.

In this study, we used Ubertooth for sniffing BLE communications, however, there are other ways to sniff BLE traffic. For instance, [10] can be used to similar to Ubertooth [11] can sniff BLE packets. Advantage of Ubertooth over Bluefruit is that it can listen Bluetooth classic packets as well as BLE, while Bluefruit can only capture BLE communications. It's also possible to capture tha Bluetooth communication on the Android device [12] while it's talking to another device over Bluetooth.

A. Gentoo GNU/Linux Installation Procedures

The target system all software will be installed on for this project is Gentoo GNU/Linux using OpenRC as the init system running on a Thinkpad T420 laptop with a built in Bluetooth card. Gentoo GNU/Linux provides an extremely powerful package management system to handle the management of Bluetooth support for various programs and drivers. Gentoo is what is known as a source-based distribution, meaning that packages are downloaded from the Gentoo repository as source code and compiled locally. While compile times may take upwards of hours for certain programs this provides a software environment tailored exactly to the abilities of the hardware it runs on. Before installing any Ubertooth specific software the BlueZ Bluetooth driver must be installed. Utilizing the Gentoo GNU/Linux wiki guide for installing Bluetooth drivers the following steps were performed:

1. Edit the /etc/portage/make.conf file to add Bluetooth as a global USE flag.

Adding a USE flag to the make.conf file tells the Gentoo system that we need our system to support Bluetooth now. However changes will not be immediately put to affect until we perform an @world update.

2. Update the world system to search for files that need to be recompiled with Bluetooth support using:

emerge –changed-use –deep @world
This command performs a deep check on the entire system to ensure that all programs with the ability to do so are recompiled according to our new USE flag, Bluetooth.

3. Install the BlueZ package with:
emerge –noreplace net-wireless/bluez

4. If a user account needs access to Bluetooth it must be added to the plugdev group using the following command. For the scope of this project however this step is unecessary as Ubertooth requires root access to function anyways, but it is nice to have this option available for other projects or just regular Bluetooth usage.
gpasswd -a ⟨user⟩ plugdev

5. To start the Bluetooth service and add it to the default runlevel use the commands:
rc-service Bluetooth start
rc-update add Bluetooth default

Now the BlueZ service will not only currently run, but will enable itself on reboot. Once Bluetooth has been successfully installed it is now time to install the Ubertooth drivers and interface program using:
emerge dev-wireless/ubertooth

For my specific Ubertooth One device the firmware was out of date. Using the instructions on the GitHub page for the project the following steps rectified the problem:

1. Clone the latest Ubertooth package from their github repository with:
$ git clone https://github.com/greatscottgadgets/ubertooth.git

2. Change directory to ubertooth-one-firmware-bin and run the ubertooth one updater tool
$ cd ubertooth-2017-03-R2/ubertooth-one-firmware-bin
$ ubertooth-dfu -d Bluetooth_rxtx.dfu -r

3. Select the ubertooth device. It will now enter DFU mode and flash the device. To exit DFU mode and resume normal operation of the Ubertooth device, simply unplug it and plug it back in.

4. To verify that the firmware has been properly updated run:
$ ubertooth-util -v
The result of this command should now be the same version of the firmware that was flashed to the device. Ubertooth One should now have everything it needs to begin work.

B. Scanning connections with Ubertooth One

Ubertooth supports a wide variety of options for capturing and scanning in various ways. First we must scan the area to find if there are any Bluetooth Low Energy devices to capture traffic from. Scanning with Ubertooth uses both the device itself and the Bluetooth device already on board the system. The following command will activate the on board device:

hciconfig hci0 up

From here we can execute the command to scan for all devices. This scan works by first listening to advertisement packets from visible devices with the on board chip then sniffs for powered on but invisible devices utilizing the Ubertooth One device. It is very unlikely that sniffing for invisible devices will reveal all the information about a device. It will typically pick up the device name, and the last three octets of the hardware address. Run the scan with the following command:
ubertooth-scan -s

Now that we have an idea of what devices are around us, we can sniff for Bluetooth Low-Energy traffic using the ubertooth-btle command. This command contains a wide variety of flags for specific modes.

These are the more common flags experimented with in this project:

-c ⟨output.pcap⟩ : This flag will be appended to all scans to record scans to the pcap file format compatible with the other programs in use for this project.

-f : Follows connections if they are established while scanning is happening. This is the recommended mode of operation.
-p : Promiscuous mode sniffs out already established connections. This mode is experimental and not recommended.

-i : When paired with -f/-p will attempt to interfere with a single connection and return to an idle state.
-I : When paired with -f/-p will attempt to interfere with all the connections it can.
Both of these flags function as an easy albiet not guarenteed to work denial of service attack on devices in within range. These flags can be leveraged to force the devices to un-pair and re-pair so the pairing event can be captured.

To sniff Bluetooth Low Energy connections and record it to a file format able to be read by the following programs, the following command was used:
ubertooth-btle -f -c out.pcap

One additional option covered by this project that Ubertooth supports is a graphical representation of the traffic in the area, organized by transmission strength in dBm and frequency. This gives a rough estimate as to the overall amount of Bluetooth traffic in the area for such as diagnostics for range finding and other things outside the scope of this project. The command to do this is:

ubertooth-specan-ui

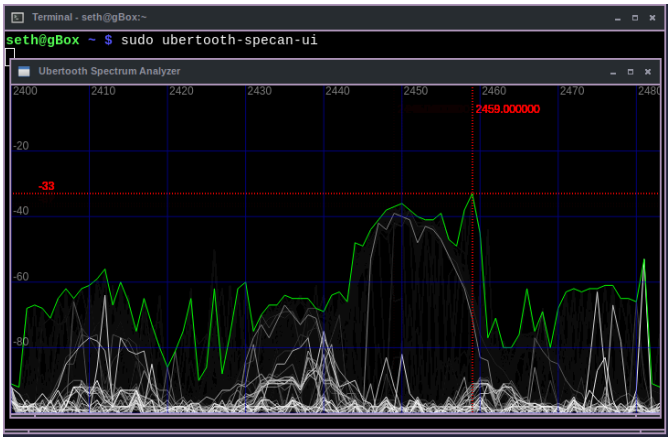

Figure 3. Ubertooth's spectrum analyzer organized by signal strength over channel frequency. White lines represent activity in the current moment while the green line represents the maximum signal strength for that frequency detected.

V. Breaking Encryption with Crackle

Raw capture data from Ubertooth cannot simply be opened and read in cleartext as-is, it must first be decrypted. Fortunately for an attacker the restricted entropy of the Temporary Key is a vulnerability. There exist two ways of attempting to decrypt a capture. First, if a scan from Ubertooth contains a complete pairing event it becomes trivial to brute-force the Temporary Key. Then once the Temporary Key has been discovered it can be applied to either the same file or additional captures of the same communication stream that do not have the initial pairing event to uncover the Long Term Key that is used to encrypt the actual data. The key can then be applied to the capture to decrypt the communication stream. Crackle supports cracking capture files either starting from the captured pairing event or with an existing TK and an incomplete stream to decrypt the LTK. Either cracking modes can return the respective keys and output the decrypted packets to a new cleartext pcap file.

A. Usage and Flags

Crackle can be run by simply calling crackle via the command line, it does not require any special privileges since (until attacking with Scapy) from this point onward all operations are done to a locally saved capture file. The following flags are options to perform the aforementioned cracks:

-i ⟨input.pcap⟩ : Specifies the encrypted file to crack
-o ⟨output.pcap⟩ : Specifies an output file for decrypting the packets after the LTK has been cracked.
-l ⟨TK⟩ : Specify a Temporary Key to apply to an incomplete capture.

Running crackle with no -i flag will return a help menu enumerating these flags and additional ones. Without a -l flag crackle will run in Crack TK mode, where it will first attempt to crack the Temporary Key then in turn decrypt the Long Term Key then the communication stream. With a -l flag crackle will run in Decrypt LTK mode where it just tries to decrypt the communication stream.

1) Packets required for these two modes: Decrypt LTK mode only requires a LL_ENC_REQ and a LL_ENC_RSP packet. Crack TK mode does the most work on the packets and so it follows that it requires the most data in the pcap file. In order to decrypt the keys it is required to have all packets in the following table present in the capture file.

Packet:	Description:
Connect Packet	Initial packet
Pairing Req	Asks to connect
Pairing Resp	Allows connection
2 Confirm values	Used for LTK
2 Random values	Used for LTK
LL_ENC_REQ	Start LTK Enc stream
LL_ENC_RSP	Allows stream

The connection starts with a connect packet from the master device, it activates the slave device and prepares it for the upcoming sequence. The Pairing request packet is then sent by the master device, and the slave device responds with a pairing response. Two confirmation value packets are exchanged, and then two random nonce values. These four numbers, alongside the Temporary Key, are used to construct the Long Term Keys on both devices. Encryption request and encryption response packets are then exchanged and the encrypted stream is now established. It is theoretically possible to brute force the Long Term Key without these data points, however this would take an unfathmoable amount of time to accomplish. Instead we capture the entire pairing event, leaving behind the Temporary Key as the only remaining variable to construct the Long Term Key. Since the Temporary Key is only six digits in length padded to 128 bits, it becomes trivial to brute force all 999,999 key possibilities.

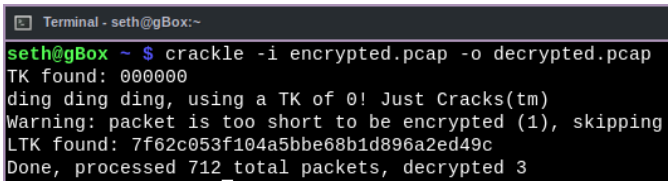

Figure 4. Crackle operating in Crack TK mode. The encryption key for this capture is just 000000, alarmingly insecure and alarmingly common.

VI. Analyzing Data with Wireshark

Wireshark is a powerful graphical packet analysis tool. It can load in standard packet capture files and apply a wide

variety of filters to sift through the mass quantities of capture data to reveal the information in question.

A. Decrypting with the LTK

Like many encryption protocols, Wireshark supports the decryption of Bluetooth encryption if a Long Term Key is given to the program for it to use. Wireshark cannot however decrypt either the Temporary Keys or Long Term Keys on its own.

B. Adaptation for Bluetooth Low Energy

In order to prepare Wireshark to understand Bluetooth Low Energy traffic on Gentoo GNU/Linux it is necessary to set the Bluetooth USE flag for net-analyzer/wireshark and re-emerge the package. On non source-based package management systems this is an unecessary step.

VII. Attacking with Scapy

Scapy is a python suite used for manually crafting packets. It supports a wide variety of protocols including the Bluetooth suite. Scapy can perform as many attacks on Bluetooth as there are Bluetooth attacks but for the scope of this project only a DDoS attack and a Replay attack. Scapy can be imported as if it were a library so functions can be called in other programs, or Scapy can be run interactively as its own shell.

A. Denial of Service Attack

Due to the low power consumption of Bluetooth Low Energy, devices that use it tend to only be able to handle small amounts of data at a time. This vulnerability can be exploited by simply sending large quantities of pairing request packets. This attack can potentially drain the battery of the device, prevent legitimate users from pairing, or even crash the device. A simple Python program that uses Scapy can be written to randomly generate source addresses and send spoofed pairing requests to the target device.

B. Replay Attack

The second attack covered by this project is the replay attack. In a single line of code Scapy can send a replay attack where the data to be sent is contained within a capture file. Given that the device is using a static key in Just Works Mode it is possible to re-send data to a device. An example of a situation where this would be a serious flaw could be a Bluetooth Low-Energy home lock system. All an attacker would need to do for a device vulnerable to this sort of attack is capture the packet stream once as the owner unlocks their home. A Bluetooth alarm device such as a Tile could be made to constantly ring to the irritation of the device owner. Tile in particular has no off switch or other way to silence the device!

C. Capturing a complete handshake

First, capturing a complete pairing event proved to be more difficult than expected. It took upwards of ten times for Ubertooth to capture all the packets needed for cracking the Temporary Key. There does not appear to be any solution to this problem other than ensuring the Ubertooth antenna is as physically close as possible to the two communicating devices. Noise from other devices in the immediate area did not seem to have much of an effect on packet loss, and using Wireshark it is straight forward to filter out the unwanted traffic.

VIII. Directions for Further Research

As demonstrated by the ambiguity of the Tile device products that claim to use Bluetooth may not properly implement the Bluetooth protocol according to standard. Naturally this will lead to new potential vulnerabilities unique to specific devices or vendors. To discover more vulnerabilities it becomes necessary to narrow down the focus of the research to individual devices. It is likely that vendors will utilize the same implementation of the Bluetooth stack across as many devices as possible, so further research could also include a range of devices from the same vendor to test if any vulnerabilities affect the vendor's entire product line.

IX. Conclusion

Bluetooth Low Energy's most severe vulnerability is it's Temporary Key. Once a handshake has been captured it becomes trivial to decrypt the following communication stream. A six digit pin is far too small to prevent a modern computer from brute forcing the key. Bluetooth Low Energy is a handy tool for small devices but it should not be used in mission critical systems or with sensitive data.

References

[1] K. Townsend, R. Davidson, and C. Cufí, *"Getting Started with Bluetooth Low Energy"*. O'Reilly, 2014. [Online]. Available: https://books.google.com/books?id=XjY4nwEACAAJ

[2] D. M. Mendez, I. Papapanagiotou, and B. Yang, "Internet of Things: Survey on Security and Privacy," *CoRR*, vol. abs/1707.01879, 2017. [Online]. Available: http://arxiv.org/abs/1707.01879

[3] Estimote, Inc., "Indoor Location with Bluetooth and Mesh," "https://estimote.com/", [Online]; accessed 29-Nov-2018.

[4] F. Zafari and I. Papapanagiotou, "Enhancing iBeacon Based Micro-Location with Particle Filtering," in *2015 IEEE Global Communications Conference (GLOBECOM)*, 12 2015, pp. 1–7.

[5] D. K. Oka, T. Furue, L. Langenhop, and T. Nishimura, "Survey of Vehicle IoT Bluetooth Devices," in *2014 IEEE 7th International Conference on Service-Oriented Computing and Applications (SOCA)*, vol. 00, Nov. 2014, pp. 260–264. [Online]. Available: doi.ieeecomputersociety.org/10.1109/SOCA.2014.20

[6] N. A. Dudhane and S. T. Pitambare, "Location Based and Contextual Services Using Bluetooth Beacons: New Way to Enhance Customer Experience," in *Lecture Notes on Information Theory*, vol. 3, no. 1, 2015, pp. 31–34.

[7] R. Bouhenguel, I. Mahgoub, and M. Ilyas, "Bluetooth Security in Wearable Computing Applications," in *2008 International Symposium on High Capacity Optical Networks and Enabling Technologies*, Nov 2008, pp. 182–186.

[8] A. Sharma, "Bluetooth Security Issues, Threats and Consequences," in *Proceedings of 2nd National Conference on Challenges & Opportunities in Information Technology (COIT-2008)*, March 2008, pp. 78–80.

[9] M. Ossmann, "Project Ubertooth," "http://ubertooth.sourceforge.net/", [Online]; accessed 29-Nov-2018.

[10] AdaFruit, Inc., "Bluefruit LE Sniffer - Bluetooth Low Energy (BLE 4.0)," "https://www.adafruit.com/product/2269", [Online]; accessed 29-Nov-2018.

[11] Tony DiCola, "Reverse Engineering a Bluetooth Low Energy Light Bulb," "https://learn.adafruit.com/reverse-engineering-a-bluetooth-low-energy-light-bulb/overview", [Online]; accessed 29-Nov-2018.

[12] M. Grassi, "How to Capture Bluetooth Packets on Android 4.4," https://www.nowsecure.com/blog/2014/02/07/bluetooth-packet-capture-on-android-4-4/, [Online]; accessed 29-Nov-2018.