Bluetooth Low Energy – Security issues

Moritz Schaefer
School of Electronic Information and
Electrical Engineering
Shanghai Jiao Tong University
Email: mollitz@gmail.com

***Abstract*—Bluetooth low energy is used in a lot of applications today. Though it's base implementation is broken by design. In this paper I will give an overview over Bluetooth low energy and its security mechanisms and then show how it can be attacked in different ways.**

***Keywords*—LaTeX, paper, security, bluetooth, ble**

I. Introduction

While the *Internet of Things* gets more and more popular, security gets a bigger and bigger concern at the same time in that topic. Many devices connected to the internet use unsafe technologies and are easily hackable over the Internet. [1]. *Bluetooth Low Energy*, or officially *Bluetooth Smart* (in the rest of the document BLE) is a quite recent technology that came up in 2010 and got a lot of attention in the meanwhile. Its powersaving capabilities allow certain devices to run with a single coin cell battery for more than a year[2]. This is a game changer and enables many new applications in the Internet of Things. Even though BLE is a modern technology, in its original specification version 4.0 the security is broken for very trivial reasons. Fortunately 5 years after this, the Bluetooth commitee improved their standard with the version number 4.2 and included a fix for the security hole[3][4]. In this paper I will show you different attacks on BLE, along with tools on how to use the attacks in practice; and give explain how the new standard version solves the security hole.

II. Wireless Technology

BLE is a standard first defined in the Bluetooth Specifications Version 4.0. It coexists with the Bluetooth Classic specification. While there are similarities (Bluetooth Low Energy is derived from Bluetooth Classic), the two standards are not compatible. A device has to implement both standards in order to communicate with devices of both kinds.

While BLE has the same top-level layers as Bluetooth Classic(GATT and L2Capp layer), it has differences on the low level layers(physical and link layer).

A. Physical layer

BLE operates in the same 2.4 GHz frequency band as Bluetooth classic. Though it uses only 40 instead of 80 channels where each channel has 2 MHz space. 37 of these channels are used for data transmission and 3 channels are advertisment channels. Devices advertise their services and the ability to connect to them on these three channels; Connection establishment is also performed on these channels. Different from Bluetooth Classic, BLE uses the Gaussian Frequency Shift Keying (GFSK) modulation scheme.

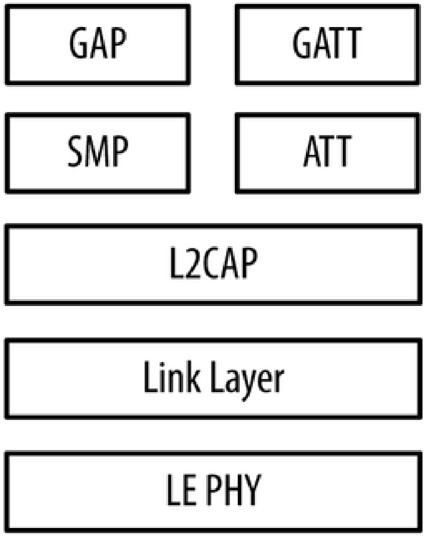

Fig. 1. Bluetooth low energy key exchange protocol

1) Channel hopping: To prevent interference with other connections and wireless technologies, BLE uses frequency hopping and changes the channel after a defined amount of time. The channels are switched according to the formulae

$$newchannel = currentchannel + hopincrement\%37$$

where advertisment channels are not considerated for simplification. The channel is changed in a defined interval hoptime, i.e. every "*hoptime*" milliseconds, the channel is hopped according to the given formulae.

B. Link layer

BLE packet are made of the components shown in figure 2 and consist of between ten and 47 bytes. The first 8 bit are a fixed preamble to indictae the packet is a BLE packet. The next 32 bit are the access address. For advertising packets, this field always has the value 0x8E89BED6 for normal data packets it is a connection specific value. The next field is the PDU(Protocol data unit) which contains the actual payload/user data and consists of 2-39 bytes dependent on the application layer packet size. In the end end there is simple CRC value which functions as a checksum to verify the integrity of the package content. In Bluetooth low energy to keep power consumption low, many mechanisms are less aggressive than on Bluetooth classic. Specificially the hopping rate and the whitening of

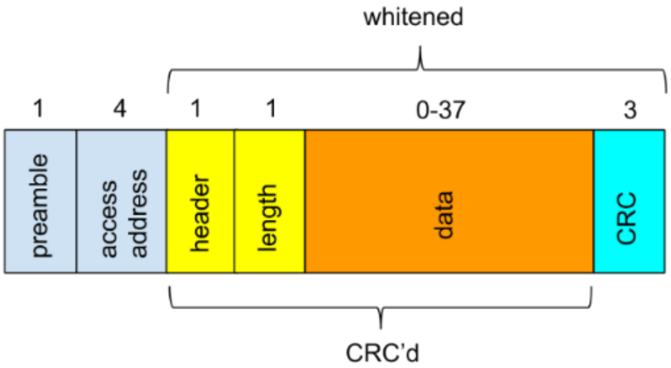

Fig. 2. Bluetooth low energy packet format [5]

PDU are simpler and as a result of that as well simpler to eavesdrop.

C. Connection

As the bluetooth peripheral device sends advertisment packages, the central device, which is often represented by a smartphone-similar device is responsible of detecting and connecting to it. During this connection establishment the two devices setup the connection which includes parameters like *channel hop time* and *channel increment*, but also initialize the encryption, if enabled. After the connection is established, the two devices talk on 37 channels with channel hopping enabled.

It is worth mentioning, that all PDU and CRC data is whitened (by XORing) with a LFSR. Though, as the seed is simply the channel number, it doesn't increase security/anonymity as it can easily be reversed in an eavesdropping process.

III. Encryption

BLE uses AES-CCM for it's encryption. This standard ensures data encryption and integrity at the same time. It is used in many applications as for example WLan and is considered to be unbroken. While this is a public and secure standard, there is nothing to worry about and analyzing this is not in the scope of the paper. Though, the key exchange protocol used by BLE is another story. The key exchange protocol was invented exclusively by the Bluetooth association and is broken by design.

A. Key exchange

As mentioned in section II-C, encryption starts during connection by exchanging the keys. This is the basis of the encryption and all subsequent keys are based on this first key exchange.

Bluetooth connecting security is established in general, by letting the two parties know the same key(the *temporary key*)and after that let them proof, that they indeed have the same key to verify them. As they cannot simply send the key to each other, as anybody could claim to have the correct key then, they compute a so called confirm value to share their knowledge about the secret without sharing the Temporary Key.

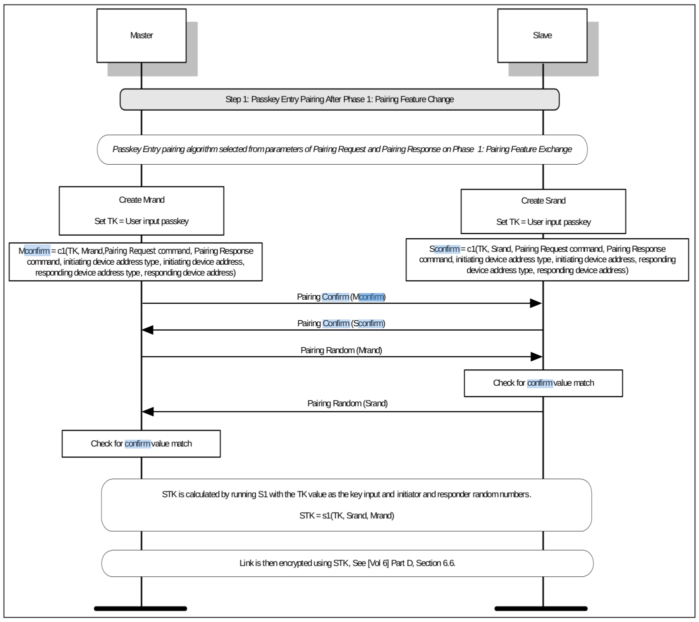
Master
Slave
Step 1: Passkey Entry Pairing After Phase 1: Pairing Feature Change
Passkey Entry pairing algorithm selected from parameters of Pairing Request and Pairing Response on Phase 1: Pairing Feature Exchange
Create Mrand
Set TK = User input passkey
Create Srand
Set TK = User input passkey
Mconfirm = c1(TK, Mrand,Pairing Request command, Pairing Response command, initiating device address type, initiating device address, responding device address type, responding device address)
Sconfirm = c1(TK, Srand, Pairing Request command, Pairing Response command, initiating device address type, initiating device address, responding device address type, responding device address)
Pairing Confirm (Mconfirm)
Pairing Confirm (Sconfirm)
Pairing Random (Mrand)
Check for confirm value match
Pairing Random (Srand)
Check for confirm value match
STK is calculated by running S1 with the TK value as the key input and initiator and responder random numbers.
STK = s1(TK, Srand, Mrand)
Link is then encrypted using STK, See [Vol 6] Part D, Section 6.6.

Fig. 3. Bluetooth low energy key exchange protocol [6]

To do so the two parties(M which starts the connection and S who responds to the connection attempt) generate one random value each ($rand_M$ and $rand_S$). Using these random values, the Temporary Key and other *known* values such as the intiating device address, based on the AES algorithm, they compute so called *confirm* values. Which are shared as shown in figure 3: M sends S his confirm value $confirm_M$ and S sends M his confirm value $confirm_S$. Then, M sends S his random value $rand_M$. S now has everything to compute $confirm_M$ and can as such verify if M is in the possession of the correct Temporary Key. If the verification succeeds, S sends his random value $rand_S$ to M so M can verify, that S has the correct Temporary Key.

After this procedure is done, both parties know, they are in the possession of the same (correct) temporary key. They compute a series of different keys based on the Temporary Key to finally be able to compute session keys used for the AES encryption of the user data.

IV. Attacks

To attack BLE it is necessary to

- detect and interpret (sniff) packages
- sniff the key exchange packets in order to crack the encryption keys

While it is always possible to eavesdrop unencrypted connections, for encrypted connections there are two scenarios in which a connection would be provably secure:

- The connection establishment, and as such the key exchange, is done in a safe, eavesdroppping secured place (for example faraday cage)
- The *OOB* pairing mode is used.

Although *OOB* pairing mode is indeed safe, it is impractical to use and is in practice never used.

A. Sniffing

To sniff bluetooth packets, a device is necessary which reads bitstreams on the given frequency band (which is one of the 2.4 GHz channels) by decoding the GFSK modulation. While this is a simple task it involves antennas, radio chips and microcontrollers. The ubertooth project [7] implements such a device enabling sniffing on the Bluetooth Low Energy physical layer adaptively (by updating the channel which should be sniffed accordingly). This bit stream may include information from many different connections and/or wireless technologies. To find a bluetooth packet in that bitstream, the *preamble* (see section II-B) can be used as a pattern to look for. The upcoming 32-bit form the access address then and show if the packet is interesting or not (is it the connection, the attacker is interested in?). Now the next bits, the PDU and CRC data, are whitenend and as such must be XORed with the an LSFR seeded by the channel number which is sniffed to clear the whitening of the data. At this point the packet is successfully restored on the link layer.

In pratice it is most convenient to sniff on the three advertisment channels: The connection establishment can be seen and all parameters necessary to follow and dercypt a connection can be recovered. As the hop-interval and hop-increment are none by the connection establishment, it is possible to change the sniffed channel according to that scheme to follow and decrypt the complete connection.

B. Connection reset

Often it is the case, that a connection has already been established and it is not possible to sniff the connection parameters in order to follow the connection. In that case it is possible to provoke a connection reset which leads to another connection establishment. To provoke a connection reset, packet injection has to be done. Looking at section II-A and II-B we see what has to be done to generate and inject a packet, given the payload, that is intended to be sent:

1) Generate PDU header
2) Generate CRC (based on PDU content)
3) Whiten data (use LSFR)
4) Generate full package by prepending access address and preamble
5) Send out on phy layer on the appropriate band using GFSK (e.g. use ubertooth)

The BLE reference shows the structure of packets to provoke disconnection and connection reset on pages 564 and 64 (Section 4.6 DISCONNECTION REQUEST (CODE 0x06)) and show which PDU data has to be used to provoke the connection reset which will lead to a new connection establishment.[6, p. 564]

C. Channel following

Section IV-B is a proper way to provoke a new connection establishment in order to sniff connection parameters which enable us to follow a communcation through all the channels. As it might be desirable to attack passively, such that no-one can detect the existent of an attacker, that attack is not ideal, as it injects packets. Though, there is a way to follow an ongoing connection by calculating the necessary hop-interval and hop-increment values, which is completely passive. Based on the fact, that the number of (data-)channels is a prime(37), we know that all channels will be used once before the first one is used again.

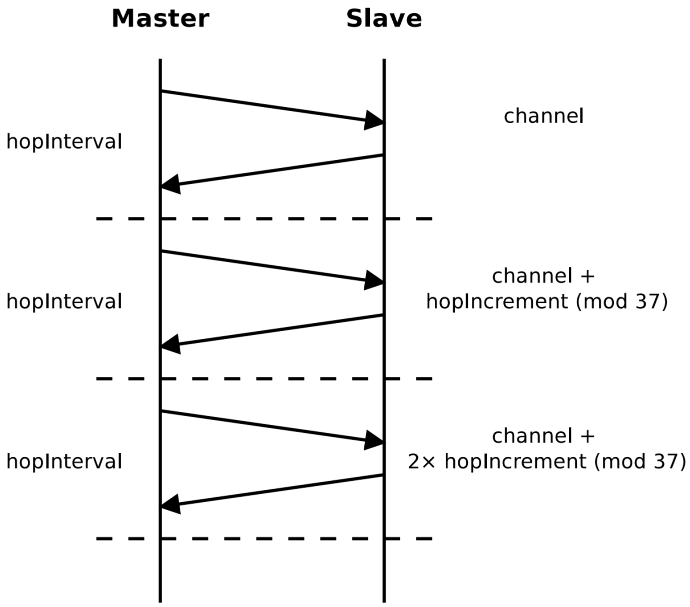

Fig. 4. BLE Hopping scheme [8]

1) Hop interval: As such we can look for an established connection (using preamble and access address) detect, when it sends packages at a certain channel once and wait for it to send data again on that channel. The time difference can be divided by 37 to get the hop-interval (the time, the connection resides on a channel before hopping to the next):

$$hopinterval = \frac{\Delta t}{37}$$

2) Hop increment: Calculating the hop-increment involves a bit more math. To gather the necessary information, we listen first on channel 0, hop to channel 1 and wait to receive a package from the desired connection. Knowing the hop-interval already, we know how many hops the connection has performed before reaching channel 1. By knowing the number of hops that were performed we can generate a lookup-table by using Fermats little theorem or simple by brute-forcing all combinations to find a mapping from hop-count to *hop increment*. Once these two values(hop interval and increment) are known, we can follow the connection in order to capture the full transmitted data.

D. Key breaking

As described in section III-A the two parties establishing a connection exchange (in plaintext) several values to ensure they have the same Temporary Key. The key in cracking the Temporary key is the confirm value.

$$
\begin{aligned}
confirm = c1(TK,\\
Mrand,\\
Pairing\\
Requestcommand,\\
PairingResponsecommand,\\
initiatingdeviceaddresstype,\\
initiatingdeviceaddress,\\
respondingdeviceaddresstype,\\
respondingdevicesaddress)
\end{aligned}
$$

c1 is a function based on AES which provides one-wayness. Looking at the equation, we can analyze which of the fields we know as an attacker (as they were transmitted in plaintext over the air). Namely:

- Confirm
- Rand
- Pairing Request command
- Pairing Respond command
- initiating device address type
- initiating device address
- responding device address type
- responding device address

In fact, the only value, we don't know in this formulae, is the temporary key TK. As told in the specification, in the most of the cases, TK is either 0 or a value between 0 and 999999. This numbers are too small to defend against a simple bruteforce attack and as such it is possible to try out all different values of TK, calculating the corresponding confirm value and compairing it with the sniffed one. Having found the TK that leads to a matching confirm value gives us the possibility to calculate the different keys defined in the specification to end up with the session key which gives us the ability to decrypt the entire transmission.

V. Key exchange protocol in 4.2

To prevent man-in-the middle attacks and passive eavesdropping as described in section IV. To do so, the specification introduces the Elliptic curve DiffieHellman (ECDH) algorithm for key generation. The algorithm enables proper asymetric encryption to share a key without caring about evesdropers.

VI. Conclusion

Bluetooth low energy in specification versions 4.0 and 4.1 is broken by design. Not only is it possible to track and eavesdrop ongoing connections, but also the key exchange protocol is broken in these versions resulting in millions of insecure connections nowadays. Tools to apply the known attacks are available in the internet. Future devices can be based on the new standard 4.2, which supports a secure key exchange protocol. Though many devices from nowadays won't get updates.

Appendix A
Proof of the prime number channel hop circle completeness

In section IV-C we refered to the fact that independently from the chosen hopincrement all channels will be used once after the same channel will be used a second time. We will proof this simple fact in this section.

To proof this we assume the opposite: There is a hopincrement (which is no multiple of 37) which reaches its start channels again, before it hopped over all 37 channels:

$$\exists x \in \mathbb{N} \setminus (\mathbb{N} * 37), \exists n \in \mathbb{N}. x * n \% 37 = 0 \land n < 37$$

which is the same as

$$\exists x \in \mathbb{N} \setminus (\mathbb{N} * 37), \exists n \in \mathbb{N} \exists y \in \mathbb{N}. x * n = 37 * y$$

$$\exists x \in \mathbb{N} \setminus (\mathbb{N} * 37), \exists n \in \mathbb{N} \exists y \in \mathbb{N}. \frac{x * n}{y} = 37$$

As we see, to get 37 on the right side either x or n must be a multiple of 37, because y is in $\mathbb{N}$ and can't be $\frac{1}{37}$. As x cannot be a multiple of 37 by definition, n has to be. As such our assumption is wrong and we can conclude:

$$\forall x \in \mathbb{N} \setminus (\mathbb{N} * 37), \forall n \in \mathbb{N}. x * n \% 37 = 0 \land n = \mathbb{N} * 37$$

This proof holds for any prime number. We just used 37 to show it specificially for the channel hopping case.

Acknowledgment

The author would like to thank Mike Ryan for his great work on Bluetooth Low Energy Sniffing and Security Research.

References

[1] G. Gang, L. Zeyong, and J. Jun, "Internet of things security analysis," in *Internet Technology and Applications (iTAP), 2011 International Conference on*, pp. 1–4, Aug 2011.

[2] M. Siekkinen, M. Hiienkari, J. Nurminen, and J. Nieminen, "How low energy is bluetooth low energy? comparative measurements with zigbee/802.15.4," in *Wireless Communications and Networking Conference Workshops (WCNCW), 2012 IEEE*, pp. 232–237, April 2012.

[3] B. SIG, "Bluetooth 4.2 core specification," tech. rep., dec 2014. https://www.bluetooth.org/en-us/specification/adopted-specifications.

[4] V. Gao, "Everything You Always Wanted to Know About Bluetooth Security in Bluetooth 4.2." http://blog.bluetooth.com/everything-you-always-wanted-to-know-about-bluetooth-security-in-bluetooth-4-2/, 2015.

[5] L. Li, "Using nRF24L01+ as A Bluetooth Low Energy Broadcaster/Beacon." http://ram.lijun.li/assets/files/wiki/img18_BLE_packet_structure.png, 2014.

[6] B. SIG, "Bluetooth 4.0 core specification," tech. rep., jun 2010. https://www.bluetooth.org/en-us/specification/adopted-specifications.

[7] M. O. et al., "Ubertooth." https://github.com/greatscottgadgets/ubertooth/, 2010–2015.

[8] M. Ryan, "Bluetooth: With low energy comes low security," in *Presented as part of the 7th USENIX Workshop on Offensive Technologies*, (Berkeley, CA), USENIX, 2013.