Bluetooth: With Low Energy comes Low Security

Mike Ryan
iSEC Partners

Abstract

We discuss our tools and techniques to monitor and inject packets in Bluetooth Low Energy. Also known as BTLE or Bluetooth Smart, it is found in recent high-end smartphones, sports devices, sensors, and will soon appear in many medical devices. We show that we can effectively render useless the encryption of any Bluetooth Low Energy link.

1 Introduction

Bluetooth Low Energy, also known as BTLE or Bluetooth Smart, is a new modulation mode and link layer packet format targeting low power embedded devices. It is typically found in recent high-end smartphones, sports devices, various sensors, and will soon appear in many medical devices. Given that the target devices for BTLE are expected to have low computation capabilities, compromises were made to simplify the protocol. Unfortunately, these decisions also undermine the privacy of the data transmitted over BTLE.

After giving a brief overview of BTLE and the Ubertooth platform, we will demonstrate how to perform eavesdropping on a BTLE device. Following that we cover packet injection and breaking the encryption of Bluetooth Low Energy.

Along with this whitepaper, we release open source tools to perform all the demonstrated attacks. Although commercial tools exist for following BTLE connections as they are established, they are designed to be used as a debugging aid and only print data values exchanged during this period. Our open source tools exceed these capabilities significantly. In addition to following new connections, we can also follow pre-existing connections by recovering connection parameters through novel techniques. We have also successfully demonstrated packet injection.

We implement a BTLE monitor on the Ubertooth platform. Leveraging the power of the platform we are able to obtain the parameters required to recover encryption keys by using brute force search over a very small keyspace.

2 Bluetooth Low Energy

Bluetooth is a short range connectivity protocol used in 9 billion devices. The number of devices integrating BTLE is expected to grow by 2.9 billion devices per year by 2016 [2].

Bluetooth Low Energy, defined in the Bluetooth Core Spec 4.0 [4], is a wireless protocol operating in the unlicensed 2.4 GHz band. While it operates in the same frequency range as other Bluetooth technologies, its operation at the PHY and link layers is incompatible. At the PHY layer BTLE uses Gaussian Frequency Shift Keying (GFSK) with a 250 kHz offset. It transmits on one of 40 channels at 1 Mbit/sec.

BTLE splits the 2.4 GHz spectrum into 40 channels spaced 2 MHz apart. 37 of the channels (data channels) are used during connections to transmit data and the remaining 3 (advertising channels) are used by unconnected masters and slaves to broadcast device information and establish connections.

Every packet begins with an 8 bit preamble, an alternating binary sequence. This is followed by a 32 bit access address (AA) which can be thought of as a unique identifier which defines a particular connection. When a device (master or slave) transmits on an advertising channel it uses a fixed value of `0x8e89bed6` as the access address. The value used on data channels is communicated by the master to the slave during connection setup. Following the 32 bit access address is a variable length Protocol Data Unit (PDU) which contains the message payload. Finally all packets end with a 24 bit CRC.

BTLE is aimed at lower-capability devices with limited power requirements such as embedded sensors. The timing parameters, specifically channel hopping rate, are

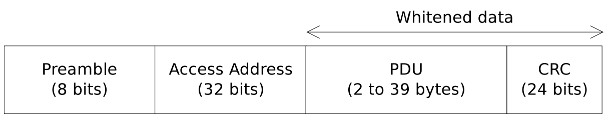

Figure 1: Bluetooth Low Energy packet format

less aggressive than other Bluetooth technologies. Other aspects of the protocol, such as whitening seed, are also simplified. These design simplifications ease the task of creating an eavesdropping tool. In addition, significant compromises were made in the key exchange protocol to account for the limited input and computing capabilities of low-power devices. While understandable in the context of the devices' constraints, these compromises undermine the privacy of the system.

3 Eavesdropping

We have implemented a sniffer capable of following BTLE connections as they hop across channels. Like commercial devices on the market [1], we are able to do so if we witness the initiation of a connection. Our major contribution is the ability to derive the parameters needed to follow a connection that has previously been established, for which we have not witnessed a connection setup.

Several major technical hurdles prevent the simplistic eavesdropping common to 802.11. First, as noted, BTLE devices hop across many channels in the 2.4 GHz spectrum, only staying on a particular channel long enough to transmit and receive a single packet. The time spent on each channel and the channel hop sequence varies from connection to connection. In addition, we rely on the 32 bit access address to determine when a packet has been transmitted, a value which also varies from connection to connection. Finally, in order to filter out false-positive packets we must verify the CRC on candidate packets, a calculation which depends on a 24 bit value known as CrcInit, which again is connection-specific.

In summary, to sniff a connection we need to know four values unique to that connection:

1. Hop interval (also referred to as dwell time)
2. Hop increment
3. Access address
4. CRC init

It is also worth noting that all data transmitted is whitened by XORing it with the output of a linear-feedback shift register (LFSR). Unlike classic Bluetooth the seed of the LFSR depends only on the channel number. In practice whitening does not complicate sniffing as the seed and LFSR are known.

3.1 Ubertooth

We built our sniffer on the Ubertooth platform. Ubertooth [6] is a USB dongle with an RF frontend, CC2400 radio chip, and LPC microcontroller. The CC2400 has a reconfigurable narrowband radio transceiver that can monitor a single BTLE channel at any given moment. The CC2400 (roughly) converts RF into a bitstream, which is then processed entirely on the LPC.

The Ubertooth project also implements a partial sniffer for classic Bluetooth. Because BTLE is a simpler protocol than classic Bluetooth, we can process packets entirely on the LPC (on-dongle). In contrast the classic Bluetooth sniffer only uses the LPC to shovel bits from the CC2400 to the PC. Our approach allows us to operate with greater agility and enables the precise timing necessary for recovering hop interval and hop increment.

Our approach also differs from the tactic used in [8] which uses a wide-band USRP to sniff several channels at the same time. We use a narrowband sniffer that is only able to tune to a single BTLE channel at any given moment. On one hand, our approach has much tighter timing requirements. On the other hand our hardware platform is much less expensive.

3.2 From RF to bytes

When a BTLE device transmits a packet on a particular channel it generates a small amount of RF energy. At the lowest level this modulated RF is what we aim to sniff. Our first order of business is to therefore convert this RF into something we can work with: bits.

We use the CC2400 radio chip on the Ubertooth to demodulate the signal. The CC2400 contains a reconfigurable modem whose demodulation parameters we configure to match those of BTLE. Namely, we configure the modem to demodulate GFSK with a frequency offset of 250 kHz and a data rate of 1 Mbit. We configure the CC2400 to be in unbuffered mode and do all bitstream processing on the LPC. We do not configure the chip to look for a preamble, though that is a future optimization worth exploring. Instead it constantly spews bits to the LPC which we process in software.

We identify the start of a transmission by searching for a known 32 bit access address. While we are on an advertising channel the AA is fixed as `0x8e89bed6`. The AA used on a data channel is exchanged during connection setup which we obtain either by sniffing the connection setup or recover using techniques described later.

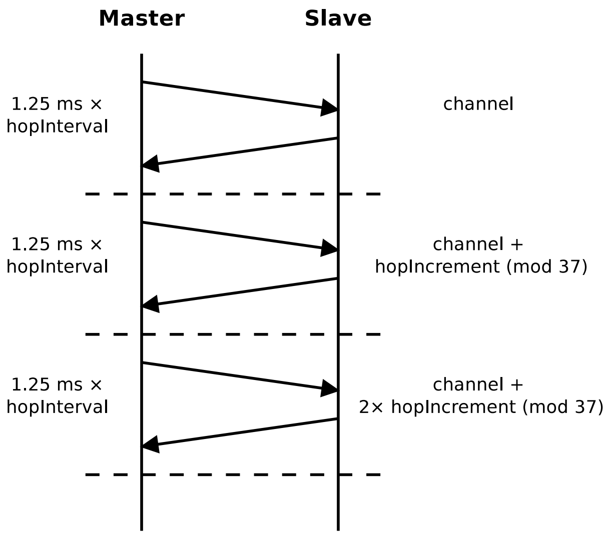

Figure 2: Master and slave each transmit on every channel, even if they have no meaningful data to exchange

The start of transmission, identified by access address, implicitly defines the byte boundary of the message. We therefore convert the bits into a sequence of bytes. From here on out, interpreting the bytes is a matter of referring to the spec.

3.3 Following Connections

We are able to convert RF to bytes on a given channel. Bluetooth Low Energy connections do not remain on a fixed channel: they hop across a sequence of data channels following a predefined hopping pattern. In order to follow a BTLE connection we must therefore hop along the same channels as the master and slave.

The BTLE channel hopping sequence is very straightforward. There are 37 data channels, and most connections use all 37. Given a *hopIncrement*, defined on a per-connection basis, the next channel in a hopping sequence is calculated by:

$$nextChannel \equiv channel + hopIncrement \pmod{37}$$

While hopping, a master and slave hop to the same channel at the same time. The master transmits a packet, and the slave transmits a packet shortly thereafter. If they have no meaningful content to exchange, they will transmit an empty data packet which consists of a header, empty body, and 24 bit CRC. The master and slave will then wait for a period time called the hop interval before hopping to the next channel as defined above. Refer to figure 2 for a graphical explanation.

In order to sniff these exchanges, our sniffer hops along the same sequence of channels at the same rate as the master and slave.

4 Promiscuous Mode

We operate on the assumption that connections use all 37 data channels. The authors have never observed otherwise, but the specification does allow fewer to be used.

In order to follow a connection, we need to know the hop interval, hop increment, access address, and CRC init as described in section 3. In connection following mode, these values are extracted from the connection initialization packet. In promiscuous mode, we recover them by exploiting properties of BTLE packets.

4.1 Determining Access Address

Promiscuous mode begins by monitoring an arbitrary data channel looking for empty data packets. These have a predictable form and are transmitted frequently. The BTLE spec requires that a master and slave transmit a packet on each channel they hop to. Hops happen frequently (typically many times per second) and only small bursts of data are normally sent during a BTLE connection, so most of the packets transmitted are empty.

Data packets consist of a 16 bit header, 0-37 octets of payload (PDU), and a 24 bit CRC. An empty data packet consists of a 16 bit header and 24 bit CRC. Two bits of the header vary (SN and NESN, used for flow control), while the rest remain a constant zero. These packets are thus easy to identify.

We read the bitstream looking for the 16 bit header that defines an empty packet. When we identify the 16 bit pattern, we treat the prior 32 bits as a candidate access address (AA). As we are only filtering the candidate bitstream by 14 bits (when discounting for the 2 varying bits of header) we are left with many false positives. We keep a least-frequently used (LFU) table of every candidate AA we observe. After we observe a candidate AA a configurable number of times (we arbitrarily choose 5) we treat it as our target AA and can filter packets based on this value.

4.2 Recovering CRCInit

Our techniques for recovering hop increment and hop interval are sensitive to timing and are ineffective in the presence of false positives. Therefore before moving onto that portion of our attack we must filter by CRC.

Every packet carries a 24 bit CRC that is calculated over the bits of the packet (including the header). This value is calculated using a linear feedback shift register (LFSR) that is pre-seeded with a 24 bit value known as CRCInit. The CRCInit varies between connections, which presents a challenge: we are unable to verify CRCs until this 24 bit seed is recovered.

In [8] Spill and Bittau note that the LFSR used to calculate a classic Bluetooth CRC is reversible. This holds true for the BTLE CRC LFSR as well.

When we receive a candidate packet, we seed the reverse LFSR with the CRC from the air. We then run the bits through the LFSR in the reverse order. The value left in the LFSR at the end of this exercise is our candidate CRCInit. As in the case of candidate access addresses, we maintain an LFU table of candidate CRCInit values and treat a value as our true CRCInit after it is observed a fixed number of times (again arbitrarily chosen to be 5 times).

4.3 Hop Interval

The hop interval is recovered by observing that the hop sequence completes a full cycle once every $37 \times 1.25 \times$ *hopInterval* milliseconds. We sit on a data channel and measure the time between two consecutive packets. We directly calculate the hop interval using this formula:

$$hopInterval = \frac{\Delta t}{37 \times 1.25 \text{ ms}}$$

If our sniffer misses a packet we may inadvertently calculate an integer multiple of the true hop interval. For this reason we measure the hop interval over several consecutive packets. After the same value has been observed a fixed number of times we treat that value as our hop interval.

4.4 Hop Increment

Finally the hop increment is recovered by measuring the interarrival time of packets on two data channels (index 0 and 1). We wait for a packet on channel index 0, then jump to channel index 1 and measure the time it takes for a second packet to arrive.

From the interarrival time, we can calculate the number of channels hopped between the first and second packet:

$$channelsHopped = \frac{\Delta t}{1.25 \text{ ms} \times hopInterval}$$

We wish to find *hopIncrement*, which satisfies the following equation:

$$0 + hopIncrement \times channelsHopped \equiv 1 \pmod{37}$$

Rearranging terms, we are left with:

$$hopIncrement \equiv channelsHopped^{-1} \pmod{37}$$

The channel hopping sequence is isomorphic to $\mathbb{Z}_{37}$, a field. This means that the multiplicative inverse of *channelsHopped* is well-defined (since *channelsHopped* is non-zero). Fermat's little theorem gives the following closed form:

$$channelsHopped^{-1} \equiv channelsHopped^{37-2} \pmod{37}$$

We use a lookup table to map the 36 possible values to the hop increment.

At this point, we have all four values needed to follow a connection, and we enter connection following mode as though we observed the initial connect packet.

5 Injection

We have implemented BTLE packet injection as a proof of concept. From Ubertooth we send undirected advertising messages broadcasting the existence of a device with a user-specified MAC address. A PC running the Linux Bluetooth stack (bluez) receives these packets and lists the device during a scan for BTLE devices.

The theory of operation is similar to receiving, but all the data flow occurs in the opposite direction. On the LPC we craft an undirected advertising packet, which has a well-defined form. The *AdvA* (advertising address) is set to the user-specified MAC address, and the packet CRC is calculated. Finally we whiten the data and send it to the CC2400 to be transmitted.

We configure the CC2400 to operate in buffered mode due to quirks of the CC2400's unbuffered mode. This does not affect the proof of concept, but a more sophisticated injector will likely require the tighter timing that can be achieved using unbuffered mode.

This proof of concept paves the way for future attacks against the crypto system as well as Bluetooth stacks on devices. We discuss this further in the Future Work section.

6 Bypassing the Encryption

BTLE features encryption and in-band key exchange. Rather than relying on a well-established key exchange protocol such as one based on Elliptic Curve Diffie-Hellmann (ECDH) [3], the Bluetooth SIG invented their own key exchange protocol. We demonstrate that this key exchange protocol has fundamental weaknesses that undermine the privacy of communication against *passive* eavesdroppers.

We note that the session encryption provided by BTLE is known to be relatively secure. BTLE uses AES-CCM [9], against which there are no known practical attacks. Our attack targets the key exchange rather than the encryption itself. Our technique is similar in principle to [5] and [7] in which an offline brute force attack is mounted to recover a secret value when all other values are transmitted over the air.

Before establishing an encrypted session, a master and a slave must establish a shared secret known as a long-term key (LTK). Under typical operation, a master and slave establish an LTK once and reuse it for future sessions. Otherwise, the master and slave establish an LTK through a key exchange protocol.

The key exchange protocol begins by selecting a temporary key (TK), a 128 bit AES key whose value depends on pairing mode. The master and slave use this value to calculate a so-called "confirm" value. Aside from the TK, all values used to calculate the confirm are exchanged in plaintext over the air. The confirm value itself is also exchanged over the air in plaintext.

We exploit the fact that all values except the TK are publicly known in order to brute force the TK.

As noted, the TK value depends on pairing mode. Three pairing modes are defined: Just Works^{TM}, 6-digit PIN, and OOB. The TK is as follows:

- Just Works^{TM}: 0
- 6-digit pin: a value between 0 and $999,999$ padded to 128 bits.
- OOB: a 128 bit value exchanged out-of-band

We use a simplistic brute force algorithm to guess TK: we calculate the confirm for every possible TK value between 0 and $999,999$. If the master and slave used Just Works^{TM} or 6-digit PIN, we will quickly find the proper TK whose confirm matches the value exchanged over the air.

In practice we find that a TK can be cracked in less than one second on a single core of an Intel Core i7 CPU. This figure could be improved by brute forcing in parallel and/or using processor-specific AES extensions.

After the confirm is calculated, the master and slave follow the rest of the key exchange protocol to establish a short-term key (STK) and finally an LTK. The STK exchange messages are encrypted using the TK, whose value we have trivially brute forced. Therefore, if we can crack the TK then we are able to decrypt the STK exchange and recover the STK. Finally the STK is used to establish a link-layer encrypted session over which the LTK is exchanged. If we crack the TK and recover the STK, we can decrypt this session and recover the LTK.

From here on out if this master and slave communicate in the future they will use the LTK that was established using the mechanism described above. This optimization means that a passive eavesdropper who is able to recover the LTK is able to decrypt any future conversation between this master and slave, rendering the in-protocol encryption next to useless.

Note that our technique is ineffective against a well-chosen OOB key. In practice we expect that Just Works^{TM} and 6-digit PIN will be used in the overwhelming majority of use-cases. Exchanging a 128 bit OOB key is cumbersome and may require specialized hardware, whereas Just Works^{TM} and 6-digit PIN are easy to implement even on the most constrained devices. To date, we have not found any devices that implement OOB key exchange.

We also note that our attack can be performed offline. A passive eavesdropper can record the key exchange and encrypted session setup to a file. An offline tool can analyze the key exchange and crack the TK. Another tool can use this information to decrypt the encrypted session and dump the LTK exchange. Future conversations that use this LTK can be recorded and decrypted offline as long as the initial encryption setup (in which the session key is established) is recorded.

We provide a tool called crackle to perform all these attacks. crackle is open source and available online. See section 10 for more information.

6.1 Mitigations and Counter-Mitigations

Certain aspects of BTLE mitigate the attacks we describe above. As noted, if the master and slave have established an LTK they need not re-establish a key using the key exchange protocol. In addition, each encrypted session uses a session-specific nonce exchanged at the beginning of the session. Therefore even if the LTK is known, if the session initialization is not captured the conversation cannot be decrypted.

We present one theoretical and one practical *active attack* against those mitigations.

To counter the first issue, we note that the BTLE protocol has provisions for a master or slave to reject a LTK. This may be used, for instance, if the slave device loses its memory. We theorize an attack in which an eavesdropper waits for an encrypted session to be initiated. At the proper moment during initialization, the eavesdropper forces a key renegotiation by injecting the appropriate link layer message (LL_REJECT_IND). We can then attack the initialization using the technique described above as if it were a new connection.

Countering the second issue, the case in which we know the LTK but do not know the session nonce, is trivial. We jam the connection, which forces the master and slave to reconnect and re-establish a secure session, allowing us to sniff the nonce. Our jammer follows along the channel hopping sequence and injects random noise (output from an LFSR). In practice this kills connections almost instantly.

7 Future Work

Our contributions demonstrate several passive attacks against the BTLE protocol. We also demonstrate a proof of concept injector on the Ubertooth platform which lays the foundation for a multitude of interesting attacks described below.

In section 6 we describe a theoretical attack to force a key renegotiation. If this attack succeeds it will prevent a master and slave from using a pre-established key to secure their communication. At this point if a master and slave wish to use encryption they must renegotiate a key, a process which we have demonstrated is vulnerable to a passive attack.

Expanding on this attack, we theorize that it is possible to perform a full man-in-the-middle (MitM) attack between the master and slave. To simplify this thought experiment, suppose we have two Ubertooth dongles connected to the same PC. On one dongle, the faux slave, we implement a slave stack that communicates with the true master. On the second dongle, the faux master, we implement a master stack that communicates with the true slave. The data would then be marshalled through the PC where it can be tampered with without detection by the target devices.

Such an attack may even be effective in the presence of encryption. We allow the true master and slave to communicate directly to establish an LTK. If the master and slave use Just Works^{TM} or 6-digit pin pairing, we can recover the LTK using the process described in section 6. We then jam the connection with one of the dongles and interpose with the faux master and slave. Since we know the LTK, we can establish independent encrypted streams between the faux and true devices that are encrypted and authenticated from the perspective of the true devices.

Fully functioning BTLE master and slave stacks also invite the possibility of a stack fuzzer. The BTLE stacks on smartphones, PCs, and slave devices all present large unexplored attack surfaces. There is great potential for memory corruption due to multiple layers of the packet (link layer and L2CAP) having variable length fields. Additionally, we expect much of the stack to be implemented in kernel space.

8 Conclusion

We presented techniques for eavesdropping on Bluetooth Low Energy conversations. We show how packets can be intercepted and reassembled into connection streams. We also demonstrate an attack against the key exchange protocol which renders the encryption useless against *passive* eavesdroppers. This eliminates any confidentially associated with the protocol.

We also provide the first BTLE sniffer that is able to follow connections that have already been established at the time of sniffing.

Finally we provide a proof of concept injector. This paves the way for many future active attacks against hosts and devices. We offer theoretical attacks for key renegotiation, man-in-the-middle, and hypothesize a BTLE stack fuzzer.

9 Acknowledgments

We thank Michael Ossmann for designing the Ubertooth hardware and for generously donating an Ubertooth dongle. We also thank Dominic Spill of the Ubertooth project. This work would not be possible without their knowledge, guidance, and humo[u]r. We also thank Mike Kershaw (dragorn), marshallh, Zero_Chaos, Jared Boone, and the rest of the Ubertooth team for technical and moral support.

Many thanks to iSEC Partners for funding a portion of this research.

Finally the author thanks his wife Jia Guo for tolerating many late nights of hacking.

10 Availability

Bluetooth Low Energy sniffing and injection is available as a part of the Ubertooth project. The project, including the Ubertooth hardware design, is open source. For source, documentation, and more information please visit:

`http://ubertooth.sourceforge.net/`

Wireshark plugins for dissecting Bluetooth Low Energy packets are available as a part of the open source libbtbb project:

`http://libbtbb.sourceforge.net/`

crackle, the BTLE encryption cracker, is available open source at:

`http://lacklustre.net/projects/crackle/`

References

[1] CC2540 USB Evaluation Module Kit. `http://www.ti.com/tool/cc2540emk-usb`.

[2] SIG Membership. `http://www.bluetooth.com/Pages/SIG-Membership.aspx`, 2013. [Online; accessed 01-May-2013].

[3] Barker, E., Johnson, D., and Smid, M. NIST SP 800-56A, Recommendation for Pair-Wise Key Establishment Schemes Using Discrete Logarithm Cryptography, Mar. 2007.

[4] Bluetooth SIG. *Bluetooth Specification Version 4.0*. Bluetooth SIG, 2010.

[5] LINDELL, A. Y. Attacks on the pairing protocol of bluetooth v2.1. In *BlackHat Briefings* (Las Vegas, NV, USA, June 2008).

[6] OSSMANN, M., AND SPILL, D. Building an All-Channel Bluetooth Monitor. In *ShmooCon 5* (Washington, DC, USA, 2009).

[7] SHAKED, Y., AND WOOL, A. Cracking the bluetooth pin. In *Proc. 3rd USENIX/ACM Conf. Mobile Systems, Applications, and Services (MobiSys)* (2005), pp. 39–50.

[8] SPILL, D., AND BITTAU, A. Bluesniff: Eve meets alice and bluetooth. In *Proceedings of the first USENIX Workshop on Offensive Technologies* (Boston, MA, USA, 2007), WOOT '07, USENIX Association, pp. 5:1–5:10.

[9] WHITING, D., HOUSLEY, R., AND FERGUSON, N. Counter with CBC-MAC (CCM). RFC 3610 (Informational), Sept. 2003.