Low Entropy Key Negotiation Attacks on Bluetooth and Bluetooth Low Energy

Daniele Antonioli
SUTD
antonioli.daniele@gmail.com

Nils Ole Tippenhauer
CISPA
tippenhauer@cispa.saarland

Kasper Rasmussen
University of Oxford
kasper.rasmussen@cs.ox.ac.uk

***Abstract*—The specification of Bluetooth and Bluetooth Low Energy includes dedicated encryption key negotiation protocols used by two parties to agree on the entropy of encryption keys. In this work, we show that an attacker can manipulate the entropy negotiation of Bluetooth and Bluetooth Low Energy to drastically reduce the encryption key space. We call our attack the Key Negotiation Of Bluetooth (KNOB) attack.**

In the case of Bluetooth, we demonstrate that the entropy can be reduced from 16 to 1 Byte. Such low entropy enables the attacker to easily brute force the negotiated encryption keys, decrypt the eavesdropped ciphertext, and inject valid encrypted messages in real-time. For Bluetooth Low Energy, we show that the entropy can still be downgraded from 16 to 7 Bytes, which reduces the attacker's effort to brute force the keys.

We implement and evaluate the KNOB attack on more than 17 Bluetooth chips (e.g., Intel Broadcom, Apple, and Qualcomm) and 15 Bluetooth Low Energy devices (e.g., Lenovo, Garmin, Samsung, Xiaomi, and Fitbit). Our results demonstrate that all tested devices are vulnerable to the KNOB attack. We discuss legacy and non-legacy compliant countermeasures to neutralize or mitigate the KNOB attack.

I. Introduction

Bluetooth BR/EDR, referred in the rest of the paper as *Bluetooth*, and *Bluetooth Low Energy* (BLE) are two widespread wireless personal area network (WPAN) technologies used by many devices such as smartphones, laptops, tablets, smartwatches, cameras, thermostats, and cars. Despite being specified in the same standard [8], Bluetooth and BLE are different (incompatible) technologies, e.g., they use different physical layers, link layers, and security architectures.

The standard for Bluetooth and BLE specifies a number of authentication, confidentially, and integrity mechanisms for Bluetooth [8, p. 1646] and for BLE [8, p. 2289]. The security and privacy of Bluetooth has been attacked and fixed several times, going all the way back to Bluetooth v1.0 [22], [46]. Several successful attacks on the (secure simple) pairing [41], [17], [6] has resulted in substantial revisions of the standard. Attacks on Android, iOS, Windows, and Linux implementations of Bluetooth were discussed in [4]. BLE legacy security mechanisms were successfully attacked [40], and while privacy mechanism were improved [13], several implementations have been attacked [3].

Bluetooth and BLE include dedicated *encryption key negotiation protocols*, that are used by two connecting devices to negotiate the entropy of the shared encryption key. Little attention has been given to the security of these protocols, that were introduced to cope with international encryption regulations, and to facilitate security upgrades [8, p. 1650]. To the best of our knowledge, all versions of the standard (including v5.0) *require* to use entropy values between 1 and 16 bytes for Bluetooth [8, p. 1650] and between 7 and 16 bytes for BLE [8, p. 2311]. The specification of both protocols does not mandate the use of integrity protection and encryption to negotiate entropy values.

Bluetooth's encryption key negotiation protocol is run every time two (already paired) devices want to establish a new connection. The protocol is implemented in the firmware of the devices' radio chip. In particular, the initiator proposes an entropy value N (an integer between 1 and 16), the other party either accepts N, or proposes an integer value lower than N, or aborts the protocol. If the other party proposes a value lower than N, e.g., $N-1$, then the initiator either accepts it, or proposes a lower value, or it aborts the protocol. In contrast, BLE's encryption key negotiation is only performed once, while the devices are pairing, and it is implemented by the devices' OS. The initiator proposes N is the pairing request message, the responder proposes N in the pairing response message, and the lowest (yet standard compliant) value is chosen.

In this paper we describe, implement, and evaluate attacks capable of making two (or more) victims use a Bluetooth encryption key with 1 byte of entropy, and a BLE encryption key with 7 bytes of entropy. Then, the attacker can brute force the encryption keys, eavesdrop and decrypt the ciphertext, and inject valid ciphertext without affecting the status of the Bluetooth or BLE link. The attacker is not required to posses any (pre-shared) secret material, and in case of Bluetooth he does not even have to observe the pairing process of the victims. The attack is effective even when the victims are using the strongest security mode (e.g., Secure Simple Pairing and Secure Connections). As a result of our attack, the attacker completely breaks the security guarantees of Bluetooth and BLE. We call our attacks **Key Negotiation Of Bluetooth (KNOB)** attacks. In [2] we proposed the KNOB attack on Bluetooth, and this paper extends the KNOB attack to Bluetooth Low Energy and it compares the two attacks.

We demonstrate how to perform a KNOB attack on a Bluetooth link, leveraging our development of several Bluetooth security procedures to generate valid keys, and the InternalBlue toolkit [30]. Then, we demonstrate the KNOB attack on a BLE link, taking advantage of our custom Linux kernel and user space stack. Finally, we present the result of our evaluation of the KNOB attack involving more than 17 vulnerable Bluetooth chips, and 15 vulnerable BLE devices. As the KNOB attack is standard-compliant, it is expected to be effective on any device implementing the specification of Bluetooth and BLE (regardless of their version).

We summarize our main contributions as follows:

- We design and implement attacks on the encryption key negotiation of Bluetooth and BLE, which we call the Key Negotiation Of Bluetooth (KNOB) attacks. Our attacks let two unaware victims negotiate and use Bluetooth encryption keys with 1 byte of entropy, and BLE encryption keys with 7 bytes of entropy. Such low entropy values can be brute forced by an attacker to recover the ciphertext and to inject valid ciphertext. As the KNOB attack is at the architectural level of Bluetooth and BLE, all standard compliant devices are potentially vulnerable to it.
- We demonstrate the practical feasibility of the KNOB attacks by implementing them for Bluetooth and BLE. Our implementations involve a man-in-the-middle attacker capable of manipulating the encryption key negotiation protocols, brute forcing the keys, and decrypting the traffic exchanged by two victims.
- We test more than 17 different Bluetooth chips, and 15 BLE devices, and find all of them to be vulnerable to the KNOB attack. We propose legacy and non legacy compliant countermeasures to mitigate and counter the KNOB attack on Bluetooth and BLE.

Our work is organized as follows: in Section II we introduce Bluetooth and Bluetooth Low Energy. We introduce the KNOB attack in Section III, and we explain the details of the attack for Bluetooth in Section IV and Section V, and for BLE in Section VI. We evaluate the KNOB attack in Section VII, and we discuss our countermeasures in Section VIII. We present the related work in Section IX. We conclude the paper in Section X.

II. Background

In this section we provide the necessary background information on Bluetooth, both BR/EDR and BLE.

A. Bluetooth

Bluetooth BR/EDR, referred in this paper as *Bluetooth*, is a widely used wireless technology for low-power short-range communications maintained by the Bluetooth Special Interest Group(SIG) [8]. Its physical layer uses the same 2.4 GHz frequency spectrum of Wi-Fi and (adaptive) frequency hopping to mitigate RF interference. A Bluetooth network is called a piconet and it uses a master-slave medium access protocol, and there is one master device per piconet. The devices are synchronized by maintaining a reference clock signal, that we indicate with CLK. Each device has a Bluetooth address (BTADD) that consists of a sequence of six bytes. The first two bytes (from left to right) are defined as non-significant address part (NAP), the third byte as upper address part (UAP), and the last three bytes as lower address part (LAP).

To establish a secure Bluetooth connection two devices first have to pair. This procedure results in the establishment of a long-term shared secret defined as *link key* (K_L). There are four types of link key: initialization, unit, combination, and master. In this paper we deal with authenticated combination link keys because they are the most secure and widely used. In particular, they are generated as part of Bluetooth Secure Simple Pairing (SSP) that is a mechanism based on Elliptic Curve Diffie Hellman (ECDH) on the P-256 curve and challenge-response based authentication.

The specification defines custom security procedures to achieve confidentiality, integrity and authentication. In the specification their names are prefixed with the letter E. In particular, a combination link key K_L is mutually authenticated by the E_1 procedure. This procedure uses a public nonce (AU_RAND) and the slave's Bluetooth address (BTADD$_S$) to generate a Signed Response (SRES) and a Authenticated Ciphering Offset (ACO). SRES is used over the air to verify that two devices actually own the same K_L. The symmetric encryption key K_C is generated using the E_3 procedure. When the link key is a combination key, E_3 uses ACO (computed by E_1) as its Ciphering Offset Number (COF), together with K_L and a public nonce (EN_RAND). E_1 and E_3 use a custom hash function defined in the specification with H. The hash function is based on SAFER+, a block cipher that was submitted as an AES candidate in 1998 [31].

There are two ways to encrypt the link-layer traffic. If both devices support Secure Connections, then encryption is performed using a modified version of AES CCM. AES CCM is an authenticate-then encrypt cipher that combines Counter mode with CBC-MAC, and it is defined in the IETF RFC 3610 [21]. If Secure Connections is not supported, then the devices use the E_0 stream cipher. The cipher is derived from the Massey-Rueppel algorithm, and it is described in the specification [8, p. 1662]. E_0 requires the Bluetooth's clock value (CLK) to synchronize the stream ciphers of the master and the slaves.

B. Bluetooth Low Energy (BLE)

BLE is a technology for wireless personal area networks (WPAN) standardized, together with Bluetooth, in the same specification [8]. BLE was introduced in 2010 (Bluetooth v4.0) to provide a technology that was simpler, and with lower power consumption than Bluetooth. BLE and Bluetooth share the same 2.4 GHz spectrum, however their implementations are incompatible (e.g., they use different physical layers and security architectures).

BLE is used to by many devices including laptops, tablets, mice, keyboards, fitness band, smartwatches, smart locks, thermostats, and smart TVs. Recently, it has been employed in other use cases such as hardware security keys [42], vehicular networks [28], secret handshakes [32], payment systems [16], and biomedical applications [48]. As all those scenarios involve the exchange of sensitive information, it is paramount that BLE provides effective security mechanisms to protect its wide attack surface.

BLE confidentially, authentication, and integrity mechanisms are specified in the Security Manager (SM) component [8, p. 2289]. Pairing, like for Bluetooth, is used to establish, and optionally authenticate, a long term secret between two devices. The specification defines the long term secret as the *Long Term Key (LTK)*. BLE specifies two types of pairings: legacy and Secure Connections. Legacy pairing uses a custom key establishment scheme, where the LTK is generated from *Short Term Key (STK)* and *Temporay Key (TK)* short term secrets. Legacy paring, unless performed with secure out of bound (OOB) data, is not considered secure against eavesdropping and man in the middle attacks, and it should not be used [40]. BLE pairing based on Secure Connections was introduced in 2014 (Bluetooth v4.2) to overcome the problems related to legacy paring. Secure Connections pairing is considered secure against eavesdropping and man in the middle attacks, as it generates the LTK using ECDH on the NIST P-256 curve, and it provides LTK's authentication.

As different BLE applications might have different security requirements, the BLE standard provides two security modes with different security levels [8, p. 2067]:

Mode 1: Authenticated encryption (AES-CCM)
- Level 1: No encryption and not authentication
- Level 2: Unauthenticated pairing with encryption
- Level 3: Authenticated pairing with encryption
- Level 4: Authenticated LE Secure Connections, and 128-bit strength encryption key.

Mode 2: Data integrity (no confidentiality)
- Level 1: Unauthenticated pairing with data signing
- Level 2: Authenticated pairing with data signing

BLE's strongest security mode (Mode 1 Level 4) uses authenticated ECDH to establish a LTK with 16 bytes of entropy, and AES-CCM to mac-then-encrypt [8, p. 2767].

C. Host Controller Interface

Modern implementations of Bluetooth and BLE provide the Host Controller Interface (HCI). This interface allows to separate each stack into two components: the host and the controller. Each component has specific responsibilities, i.e., the controller manages low-level radio and baseband operations, and the host manages high-level application layer profiles. Typically, the host is implemented in the operating system and the controller in the firmware of the Bluetooth chip. For example, BlueZ and Bluedroid implement the HCI host on Linux and Android, and the firmware of a Qualcomm or Broadcom radio implements the HCI controller. The host and the controller communicate using the Host Controller Interface (HCI) protocol. This protocol is based on commands and events, i.e., the host sends (acknowledged) commands to the controller, and the controller uses events to notify the host. The HCI protocol can use different physical transports such as UART, SPI, and USB.

III. KNOB Attack Introduction

In this section we introduce the high level details of the Key Negotiation Of Bluetooth (KNOB) attack. The technical details for Bluetooth and BLE are presented respectively in Sections IV and V, and Section VI.

A. System and Attacker Model

We assume a system composed of two or more legitimate devices that communicate using either Bluetooth or BLE (as described in Section II). Without loss of generality, we focus on a piconet with a master (Alice) and a slave (Bob), and we indicate their Bluetooth addresses with BTADD_M and BTADD_S. The victims are capable of using Secure Simple Pairing and Secure Connections. This combination enables the highest security level of Bluetooth, and should protect against eavesdropping and active man in the middle attacks. The victims are synchronized using the master Bluetooth clock, (CLK), and the clock does not provide any security guarantee.

The attacker (Charlie) wants to decrypt all messages exchanged between Alice and Bob, and inject valid encrypted messages, without being detected. We define two attacker models: a *remote* attacker and a *local* attacker. A remote attacker controls a device that is in Bluetooth range with Alice and Bob. He is able to capture public nonces, CLK, and encrypted messages, actively manipulate unencrypted communication, and to drop packets using techniques such as network man-in-the-middle and manipulation of physical-layer signals [45], [37]. The local attacker is able to compromise the firmware of the Bluetooth chip or the OS of the BLE's device (e.g.,using backdoors [10], supply-chain implants [15], and rogue chip manufacturers [38]). In both cases the attacker has no access to any Bluetooth (pre-shared) secret quantity, i.e., Bluetooth's K_L and BLE's LTK.

B. KNOB Attack

The encryption key negotiation protocols of Bluetooth and BLE serve to cope with international encryption regulations and should facilitate security upgrades [8, p. 1650]. Those protocols have three significant problems:

1) They allow to negotiate entropy values as low as 1 byte for Bluetooth, and as low as 7 bytes for BLE.
2) They are neither encrypted nor authenticated.
3) They are transparent to the end users, and for Bluetooth they are also transparent to the Bluetooth host (OS).

In this paper we demonstrate that an attacker can convince any two Bluetooth and BLE victims to negotiate the lowest possible, yet standard-compliant, encryption key entropy value. We call our attacks **Key Negotiation Of Bluetooth (KNOB)** attacks. For example, a (remote) attacker, without any knowledge of the victims' long term secret, can let two victims use a Bluetooth encryption key with 1 byte of entropy, and then brute force the key with minimal effort (i.e., find 1 key from 256 candidate keys).

The KNOB attack on Bluetooth and BLE consists of the following high level steps:

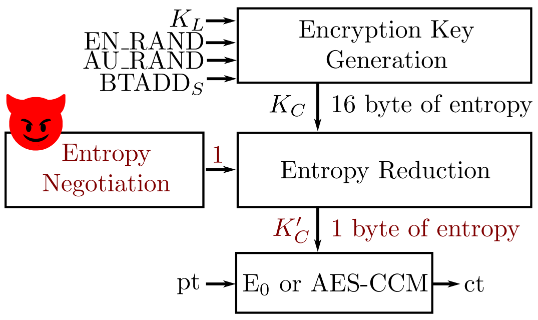

Fig. 1: Adversarial generation of a Bluetooth encryption key (K'_C). K_C is generated from K_L and other public parameters. K_C has 16 bytes of entropy, and it is not directly used as the encryption key. K'_C, the actual encryption key, is computed by reducing the entropy of K_C to N bytes. Charlie (the attacker) let Alice and Bob agree on $N = 1$, and as a result, K'_C has 1 byte of entropy. K'_C is then used for link layer encryption by either the E_0 or AES-CCM.

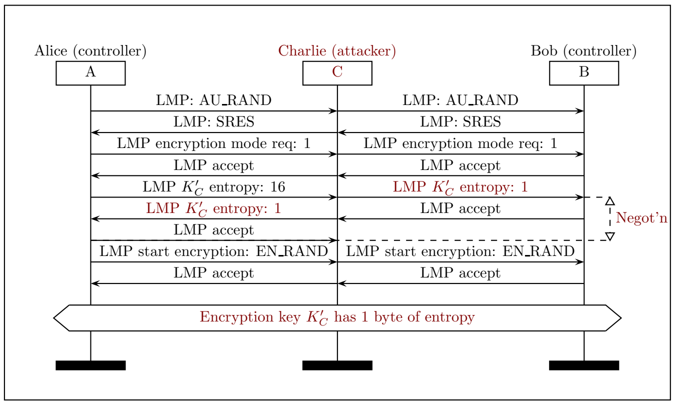

Fig. 2: Adversarial manipulation of the Bluetooth entropy negotiation protocol over LMP. The attacker (Charlie) manipulates the entropy suggested by Alice from 16 to 1 byte. Bob accepts Alice's proposal, and Charlie changes Bob's acceptance to a proposal of 1 byte. Alice, accepts the standard-compliant proposal of Bob, and Charlie drops Alice's acceptance message because Bob already accepted Alice's proposal (adversarially modified by Charlie). As a result, Alice and Bob use a K'_C with 1 byte of entropy.

1) Alice and Bob (the victims) begin to establish a connection using Bluetooth's strongest security mode
2) Charlie makes the victims negotiate an encryption key with low entropy (e.g., 1 byte for Bluetooth)
3) Charlie brute forces the encryption key using some eavesdropped ciphertext as an oracle
4) Charlie decrypts all messages and injects valid ciphertext

KNOB attacks on Bluetooth and BLE exploit their respective key negotiation protocols. The main differences between the two are summarized below:

- Bluetooth's key negotiation is performed by two already paired devices every time they want to connect, while BLE's key negotiation is performed in the first phase of pairing (i.e., the feature exchange phase).
- Bluetooth specifies that the key negotiation should be managed by the HCI controller (implemented by the firmware of the Bluetooth chip), while the key negotiation of BLE should be managed by the HCI host (implemented by the main OS of the device).
- Bluetooth's key negotiation is carried using the LMP protocol, while BLE uses the SMP protocol.
- Bluetooth's key negotiation may involve more than a request and a response packet, while BLE involves only one request and one response packet.

IV. KNOB Attack on Bluetooth

In this section we explain the technical details to conduct the KNOB attack on Bluetooth. Our implementation is discussed in Section V.

A. *Negotiate Low Entropy Bluetooth Encryption Keys*

Every time a Bluetooth connection requires link-layer encryption, Alice and Bob compute an encryption key K_C based on K_L, $BTADD_S$, AU_RAND, and EN_RAND (see top part of Figure 1). K_L is the link key established during secure simple pairing and the others parameters are public. Assuming ideal random number generation, the entropy of K_C is always 16 bytes, however K_C is *not* directly used as the encryption key for the current session. The actual encryption key, indicated with K'_C, is computed by reducing the entropy of K_C to N bytes, where N is the outcome of the Bluetooth *encryption key negotiation protocol* (Entropy Negotiation in Figure 1).

To understand how an attacker can set N equal to 1 (or to any other standard-compliant value), we have to look at the details of the encryption key negotiation protocol. The protocol is run between the Bluetooth chip of Alice and Bob using the Link Manager Protocol (LMP) protocol. LMP is neither encrypted nor authenticated, and LMP packets do not propagate to higher protocol layers. Hence, the hosts (OSes) are not aware of the LMP packets exchanged over the air by the Bluetooth controllers.

In the following, we provide an example where both Alice (the master) and Bob (the slave) are able to support an entropy value from 16 to 1 byte. The Bluetooth standard enables to set the minimum and maximum entropy values by setting two parameters defined as L_{min} and L_{max}. These values can be set and read only by the Bluetooth chip (firmware). Indeed, our scenario describes a situation where the firmwares of Alice and Bob declare $L_{max} = 16$ and $L_{min} = 1$.

Charlie's manipulation of N is described in Figure 2. The first two messages allow Alice to authenticate that Bob possesses the correct K_L. Then, with the next two messages, Alice requests to initiate Bluetooth link layer encryption and Bob accepts. Now, the negotiation of N takes place (Negot'n in Figure 2). Alice proposes 16 bytes of entropy, Charlie manipulates this value to 1 and forwards it to Bob. Bob accepts the proposal, and Charlie manipulates the message proposing to Alice 1 Byte of entropy. Alice accepts the proposal, and Charlie drops the message. As a result, of this negotiation Charlie forces Alice and Bob to use a K'_C with 1 byte of entropy ($N = 1 = L_{min}$). The Bluetooth hosts of Alice and Bob do not have access to K_C and K'_C, as they are only informed about the outcome of the negotiation. The same adversarial negotiation can be performed when the negotiation procedure is initiate by Bob (the slave).

It is reasonable to think that the victim could prevent or detect this attack using a proper value for L_{min}. However, the standard does not state how to explicitly take advantage of it, e.g., deprecate L_{min} values that are too low. The standard states the following: "The possibility of a failure in setting up a secure link is an unavoidable consequence of letting the application decide whether to accept or reject a suggested key size." [8, p. 1663]. This statement is ambiguous because it is not clear what the definition of "application" is in that sentence. As we show in Section VII, this ambiguity results in no-one being responsible for terminating connections with low entropy keys in practice. In particular, the entity who decides whether to accept or reject the entropy proposal is the firmware of the Bluetooth chip by setting L_{min} and L_{max} and managing the entropy proposals.

The "application" (intended as the Bluetooth application running on the OS using the firmware as a service) cannot check and set L_{min} and L_{max}, and it is not directly involved in the entropy acceptance/rejection choice (that is performed by the firmware). The application can interact with the firmware using the HCI protocol. In particular, it can use the HCI Read Encryption Key Size request, to check the amount of negotiated entropy *after* the Bluetooth connection is established and theoretically abort the connection. This check is neither required nor recommended by the standard as part of the key negotiation protocol.

The adversarial low entropy negotiation in Figure 2 can be performed by both attacker models that we specify in Section III-A. The remote attacker has the capabilities of dropping and injecting valid plaintext (the encryption key negotiation protocol is neither encrypted nor authenticated). The local attacker can modify few bytes in the Bluetooth firmware of a victim to always negotiate 1 byte of entropy.

B. Brute Forcing the Low Entropy Bluetooth Encryption Key

Bluetooth has two link layer encryption schemes, one is based on the E_0 cipher (legacy), and the other on the AES-CCM cipher (Secure Connections). Our KNOB attack works in both cases. In particular, if the negotiated entropy for the encryption key (K'_C) is 1 byte, then the attacker can trivially brute force it trying (in parallel) the 256 K'_C's candidates against one or more cipher texts. The attacker does not have to know what type of application layer traffic is exchanged, because a Bluetooth packet contains well known fields, such as L2CAP and RFCOMM headers, that the attacker can use as oracles.

We now describe how to compute all 1 byte entropy keys when E_0 and AES-CCM are in use. Each encryption mode involves a specific entropy reduction procedure that takes N and K_C as inputs and produces K'_C as output (Entropy Reduction in Figure 1). The specification calls this procedure Encryption Key Size Reduction [8].

$$K'_C = g_2^{(N)} \otimes \left(K_C \bmod g_1^{(N)} \right) \qquad (E_s)$$

In case of E_0, K'_C is computed using Equation (E_s), where N is the negotiated integer between 1 and 16 (see Section IV-A). $g_1^{(N)}$ is a polynomial of degree $8N$ used to reduce the entropy of K_C to N bytes. The result of the reduction is encoded with a block code $g_2^{(N)}$, a polynomial of degree less or equal to $128-8N$. The values of those polynomials depend on N, and they are tabulated in [8, p. 1668]. When $N = 1$, we compute the 256 candidate K'_C by multiplying all the possible 1 byte reductions $K_C \bmod g_1^{(1)}$ (the set `0x00...0xff`) with $g_2^{(1)}$ (that equals to `0x00e275a0abd218d4cf928b9bbf6cb08f`).

In case of AES-CCM the entropy reduction procedure is simpler than the one of E_0. In particular, the $16 - N$ least significant bytes of K_C are set to zero. For example, when $N = 1$ the 256 K'_C candidates for AES-CCM are the set `0x00...0xff`.

In the implementation of our KNOB attack brute force logic, we pre-compute the 512 keys with 1 byte of entropy and we store them in a look-up table to speed-up comparisons. More details about the brute force implementation are discussed in Section V.

C. KNOB Attack Root Causes for Bluetooth

The root causes of the KNOB attack are shared between the specification and the implementation of Bluetooth confidentially mechanisms. On one side the specification is defining a vulnerable encryption key negotiation protocol that allows devices to negotiate low entropy values. On the implementation side (see Section VII), the Bluetooth applications that we test fails to check the negotiated entropy in practice. This is understandable because they are implementing a specification that is not mandating or recommending an entropy check.

We do not see any reason to include the encryption key negotiation protocol in the specification of Bluetooth. From our experiments (presented in Section VII) we observe that two devices, unless under attack, are always negotiating 16 bytes of entropy. Furthermore, the entropy reduction performed as part of the protocol does not improve runtime performances because the size of the encryption key is fixed to 16 bytes even when its entropy is reduced.

V. KNOB Attack Implementation for Bluetooth

We now discuss how we implement the KNOB attack using a reference attack scenario. We explain how we manipulate the key negotiation protocol, brute force the encryption key (K'_C) using eavesdropped traffic, and validate K'_C by computing it from K_L (as in Figure 1). As a result, the attacker is able to decrypt the content of a link-layer encrypted file sent from a Nexus 5 to a Motorola G3 using the Bluetooth OBject EXchange (OBEX) profile. A Bluetooth profile is the equivalent

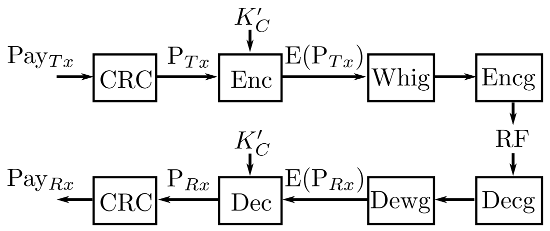

Fig. 3: Transmission and reception of an E_0 encrypted payload. The concatenation of the payload and its CRC (P$_{Tx}$) is encrypted, whitened, encoded and then transmitted. On the receiver side the steps are applied in the opposite order. RF is the radio frequency wireless channel.

of an application layer protocol in the TCP/IP stack. We use our implementation to conduct successful KNOB attacks on more than 17 different Bluetooth chips, the attacks are evaluated in Section VII.

A. Attack Scenario

To describe our implementation we use a reference attack scenario where the victims are a Nexus 5 and a Motorola G3(Table I lists their relevant specifications). We use the Nexus 5 also as a man-in-the-middle attacker by patching its Bluetooth firmware. This setup allows us to reliably *simulate* a remote man-in-the-middle attacker (more details in Section V-B).

The victims use Secure Simple Pairing to generate K_L (the link key) and authenticate the users, the entropy reduction function from Equation (E_s), and E_0 legacy encryption. The victims use legacy encryption because the Nexus 5 does not support Secure Connections. Nevertheless, the KNOB attack is effective also with Secure Connections.

Every data packet encrypted with $E_0$is transmitted and received as in Figure 3. A cyclic redundancy checksum (CRC) is computed and appended to the payload (Pay$_{Tx}$). The resulting bytes (P_{Tx}) are encrypted with E_0 using K'_C. Then, the ciphertext is whitened, encoded, and transmitted over the air. On the receiver side the following steps are applied in sequence: decoding, de-whitening, decryption, and CRC check. As E_0 is a stream cipher, the encryption and decryption procedures are the same, i.e., the same synchronized keystream is XORed with the sent plaintext and the received ciphertext. The whitening and encoding procedures do not add any security guarantee.

B. Manipulation of the Bluetooth Entropy Negotiation

We implement the manipulation of the encryption key negotiation protocol (presented in Section IV-A) by extending the functionality of InternalBlue v0.1 [30], and using it to patch the Bluetooth firmware of the Nexus 5. Our InternalBlue modifications allow to manipulate all incoming LMP messages *before* they are processed by the entropy negotiation logic, and all outgoing LMP messages *after* they've been processed by the entropy negotiation logic. The entropy negotiation logic is the code in the Bluetooth firmware that manages the encryption key negotiation protocol, and we do not modify it. As a result, we can use a Nexus 5 (or any other device supported by InternalBlue) both as a victim and a remote KNOB attacker, without having to deal with the practical issues related with packet manipulation attacks over-the-air.

InternalBlue is an open-source toolkit capable of interfacing with the firmware of the BCM4339 Bluetooth chip in Nexus 5 phones. As InternalBlue v0.1 is not providing a way to directly hook the firmware's key negotiation logic, we extend it to enable two victims (one is always the Nexus 5) to negotiate one (or more) byte of entropy. InternalBlue requires a rooted Nexus 5, and to compile and install an Android Bluetooth stack (`bluetooth.default.so`) with debugging features enabled. InternalBlue allows to patch the BCM4339 Bluetooth firmware using dedicated hooks, and read the ROM and the RAM of firmware at runtime.

Our manipulation of the entropy negotiation works regardless the role of the Nexus 5 in the piconet, and it does not require to capture any information about SSP. Assuming that the victims are already paired, we test if two victims are vulnerable to the KNOB attack as follows:

1) We connect over USB the Nexus 5 with the X1 laptop, we run our version of InternalBlue, and we activate LMP and HCI monitoring.
2) We connect and start the Ubertooth One capture over the air focusing only on the Nexus 5 piconet (using UAP and LAP flags).
3) We request a connection from the Nexus 5 to the other victim (or vice versa) to trigger the encryption key negotiation protocol over LMP.
4) Our InternalBlue patch changes the LMP packets as Charlie does in Figure 2.
5) If the victims successfully complete the protocol, then they are vulnerable to the KNOB attack, and we can decrypt the ciphertext captured with the Ubertooth One.

We now describe how we extend InternalBlue v0.1 to manipulate the LMP negotiation packets. The `internalblue/fw_5.py` file contains several information about the (reversed) BCM4339 firmware, and it provides two hooks into the firmware, defined by Mantz (the main author of InternalBlue) as `LMP_send_packet` and `LMP_dispatcher`. The former hook allows to execute code every time an LMP packet is about to be sent, and the latter whenever an LMP packet is received. The hooks are intended for LMP monitoring, and we extend them to also perform LMP manipulation.

Listing 1 shows three ARM assembly code blocks that we add to `fw_5.py` to let the Nexus 5 and the Motorola G3 negotiate arbitrary entropy values (e.g.,1 byte). This code works when the Nexus 5 is the master, and it initiates the encryption key negotiation protocol. The first block translates to: if the Nexus 5 is sending an LMP K'_C entropy proposal then change it to 1 byte. This block is executed when the Nexus 5 starts an encryption key negotiation protocol. The code allows to propose any entropy value by `mov`ing a different constant into `r2` in line 5.

		Bluetooth			
Phone	OS	Version	MAC	SC	Chip
Nexus 5	Android 6.0.1	4.1	48:59:29:01:AD:6F	No	Broadcom BCM4339
Motorola G3	Android 6.0.1	4.1	24:DA:9B:66:9F:83	Yes	Qualcomm Snapdragon 410

TABLE I: Relevant technical specifications of Nexus 5 and Motorola G3 devices used to describe our attack implementation. The SC column indicates if a device supports Secure Connections.

Listing 1 We add three ARM assembly code blocks to `internalblue/fw_5.py` to negotiate K'_C with 1 byte of entropy. In this case the Nexus 5 is the master and it initiates the encryption key negotiation protocol.

```
# Send LMP Kc' entropy 1 rather than 16
ldrb   r2, [r1]
cmp    r2, #0x20
bne    skip_sent_ksr
mov    r2, #0x01
strb   r2, [r1, #1]
skip_sent_ksr:

# Recv LMP Kc' entropy 1 rather than LMP accept
ldrb    r2, [r1]
cmp     r2, #0x06
bne     skip_recv_aksr
ldrb    r2, [r1, #1]
cmp     r2, #0x10
bne     skip_recv_aksr
mov     r2, #0x20
strb    r2, [r1]
mov     r2, #0x01
strb    r2, [r1, #1]
skip_recv_aksr:

# Send LMP_preferred rate rather than LMP accept
# Simulate an attacker dropping LMP accept
ldrb   r2, [r1]
cmp    r2, #0x06
bne    skip_send_aksr
ldrb   r2, [r1, #1]
cmp    r2, #0x10
bne    skip_send_aksr
mov    r2, #0x48
strb   r2, [r1]
mov    r2, #0x70
strb   r2, [r1, #1]
skip_send_aksr:
```

The second block from Listing 1 translates to: if the Nexus 5 is receiving an LMP accept (entropy proposal), then change it to an LMP K'_C entropy proposal of 1 byte. This code is used to let the Nexus 5 firmware believe that the other victim proposed 1 byte, while she already accepted 1 byte. The third blocks translates to: if the Nexus 5 is sending an LMP accept (entropy proposal), then change it to an LMP preferred rate. This allows to obtain the same result of dropping an LMP accept packet, because the LMP preferred rate packet does not affect the state of the encryption key negotiation protocols.

We develop and use similar patches to cover all the attack cases: Nexus 5 is the master and does not initiate the connection, Nexus 5 is the slave and initiates the connection, and Nexus 5 is the slave and does not initiate the connection.

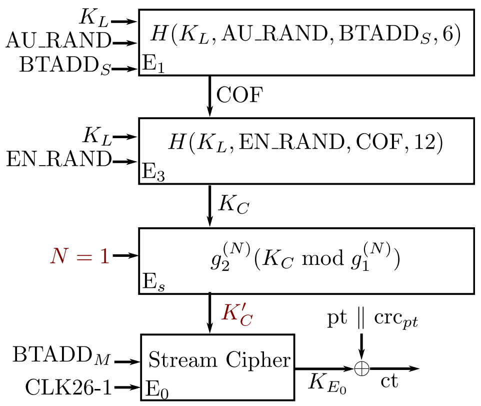

Fig. 4: Implementation of the KNOB attack on the E_0 cipher. The attacker makes the victims agree on a K'_C with one byte of entropy ($N = 1$) and then brute force K'_C, without knowing K_L and K_C.

C. Brute Forcing the Encryption Key

Once the attacker is able to reduce the entropy of the encryption key (K'_C) to 1 byte, he has to brute force it (key space is 256). In this section we explain how we brute force, validate, and use a K'_C with 1 byte of entropy (E_0 encryption).

The details about the E_0 encryption scheme are presented in Figure 4, we describe them backwards starting from the E_0 cipher. E_0 takes three inputs: BTADD$_M$, CLK26-1, and K'_C. CLK26-1 are the 26 bits of CLK in the interval CLK[25:1] (assuming that CLK stores its least significant bit at CLK[0]). The BTADD$_M$ is the Bluetooth address of the master, and it is a public parameter.

For E_0 we use an open-source implementation [11], that we correctly verify first against the specification. To provide valid K'_C candidates to E_0 we implement the E_s entropy reduction procedure, that takes an input with 16 bytes of entropy (K_C) and computes an output with N bytes of entropy (K'_C). E_s involves modular arithmetic over polynomials in Galois fields, and we use the BitVector [24] Python module to perform such computations.

We use a Python brute force script to compute the correct K'_C by testing the decryption of one (or more) ciphertext against the 256 K'_C's candidates. We validate our brute force script by decrypting the content of a file (in ASCII `aaaabbbbccccdddd`) sent from the Nexus 5 to the Motorola G3 using the OBEX Bluetooth profile, after the negotiation of a K'_C with 1 byte of entropy.

We verify that the brute forced K'_C is effectively the one in

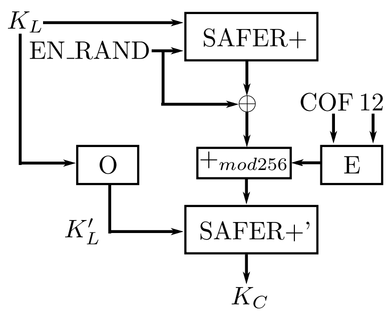

Fig. 5: Bluetooth defines H a custom hash function based on SAFER+ and SAFER+'. H is used to compute K_C from K_L, EN_RAND, and COF (see Equation E_3).

Name	Value
Public	
BTADD_M	`0xccfa0070dcb6`
BTADD_S	`0x829f669bda24`
AU_RAND	`0x722e6ecd32ed43b7f3cdbdc2100ff6e0`
EN_RAND	`0xd72fb4217dcdc3145056ba488bea9076`
SRES	`0xb0a3f41f`
N	`0x1`
Secret	
K_L	`0xd5f20744c05d08601d28fa1dd79cdc27`
COF=ACO	`0x1ce4f9426dc2bc110472d68e`
K_C	`0xa3fccef22ad2232c7acb01e9b9ed6727`
K'_C	`0x7fffffffffffffffffffffffffffffff`

TABLE II: Public and secret values (in hexadecimal) collected during a KNOB attack involving authenticated SSP and E_0 encryption. The encryption key (K'_C) has 1 byte of entropy.

use, by implementing E_1 and E_3, and use them to compute K'_C as a victim. In particular, we use the necessary parameters from Figure 4 to compute K'_C from K_L. We capture those parameters using the Bluetooth logging capabilities offered by Android. Table II shows an example of actual public and private values from a KNOB attack.

E_1 computes the Ciphering Offset Number (COF), while E_3 computes K_C (see Figure 4). Both procedures use a custom hash function defined in the specification with H. We write E_1 and E_3 equations and label them with their respective names as follows:

$$SRES \| ACO = H(K_L, \text{AU\_RAND}, \text{BTADD}_S, 6) \quad (E_1)$$

$$K_C = H(K_L, \text{EN\_RAND}, \text{COF}, 12) \quad (E_3)$$

Figure 5 shows how E_3 uses the H hash function. H internally uses SAFER+, a block cipher submitted as an AES candidate in 1998 [31] and SAFER+' (SAFER+ prime). The specification indicates them with A_r and A_r' [8, p. 1676]. SAFER+' is a modified version of SAFER+ such that the input of the first round is added to the input of the third round. This modification was introduced in the specification to avoid SAFER+' being used for encryption [8, p. 1677]. The specification uses SAFER+ and SAFER+' with 128 bit block size (8 rounds), in ECB mode, and only for encryption.

We implement in Python both SAFER+ and SAFER+' including the round computations and the key scheduling algorithm. We test the two against the specification. We also implement in Python the E and O blocks (see Figure 5), and we test them against the specification. The E block is an extension block that transforms the 12 byte COF into a 16 byte sequence using modular arithmetic. The E block is also applied to the 6 byte BTADD_S in E_1. The O block is offsetting K_L using algebraic (modular) operations and the largest primes below 257 for which 10 is a primitive root. Finally, with all code blocks in place, we implement H, and use it to implement and test E_3 and E_1. We plan to release our code implementing E_s, E_1, and E_3 as open-source[1].

[1]See https://github.com/francozappa/knob

D. Implementation for Bluetooth Secure Connections

The specification allows to perform the KNOB attack even when the victims are using Secure Connections. We implement the entropy reduction function of the brute force script over AES–CCM. However, InternalBlue v0.1 is not capable of patching the firmware of a Bluetooth chip that supports Secure Connections, indeed we are not able to implement the low entropy negotiation part of the attack using InternalBlue.

VI. KNOB Attack on Bluetooth Low Energy

In this section we explain how to conduct the KNOB attack on BLE, regardless its security mode. The details of the KNOB attack on Bluetooth are presented in Section IV and Section V.

A. Negotiate Low Entropy BLE Encryption Keys

The encryption key negotiation of BLE is part of its pairing process. BLEpairing has three phases feature exchange (that includes key negotiation), key establishment and optional authentication, and key distribution (over encrypted link) [8, p. 2296]. BLE key negotiation allows entropy values between 7 and 16 bytes, while Bluetooth allows values between 1 and 16 bytes.

As the BLE specification does not require to protect the integrity of the feature exchange phase, that includes key negotiation, we are capable of reducing the entropy of any BLE encryption key (LTK) to 7 bytes, regardless the usage of authenticated Secure Connections. The KNOB attack on BLE is conceptually similar to one on Bluetooth (introduced in Section IV), with the exception that, in case of BLE, the attacker can downgrade the key's entropy as low as 7 bytes, rather than 1 byte.

In Figure 6, we demonstrate how Charlie (the attacker) can manipulate the encryption key negotiation of a BLE connection between Alice and Bob. Alice (the master) always initiates the pairing session by sending a pairing request to Bob (the slave). The pairing request contains Alice's capabilities including input-output (IO), authentication requirements (AuthReq), and proposed encryption key size (KeySize).

In Figure 6, Charlie manipulates Alice's paring request by changing her proposed key size (KeySize) from 16 to 7 bytes

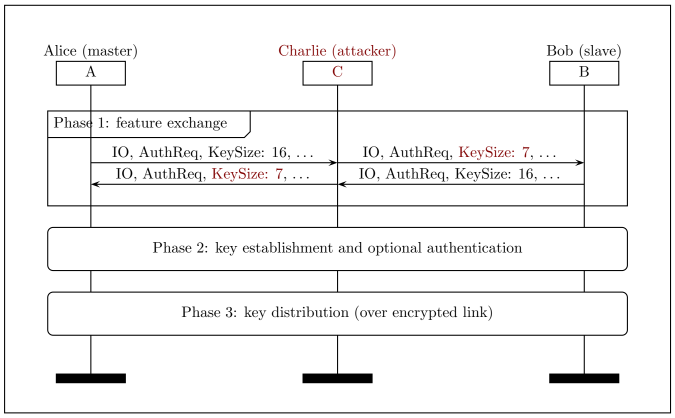
Alice (master)
Charlie (attacker)
Bob (slave)
A
C
B
Phase 1: feature exchange
IO, AuthReq, KeySize: 16, . . .
IO, AuthReq, KeySize: 7, . . .
IO, AuthReq, KeySize: 7, . . .
IO, AuthReq, KeySize: 16, . . .
Phase 2: key establishment and optional authentication
Phase 3: key distribution (over encrypted link)

Fig. 6: Adversarial manipulation of the BLE feature exchange phase. Charlie let Alice and Bob negotiate a LTK with 7 bytes of entropy (KeySize: 7). As the feature exchange phase is not integrity protected, the attacker can spoof all the capabilities declared by Alice and Bob including LTK's entropy.

(i.e., from the maximum to the minimum value allowed by the BLE specification). Then, Bob answers with a pairing response containing his capabilities, including his KeySize, and Charlie changes it from 16 to 7 bytes. As a result, the LTK (and all the secrets computed from it) have 7 bytes of entropy.

Additionally, Charlie, using the same technique described in Figure 6, can adversarially manipulate other capabilities declared by the victims during the feature exchange phase. For example, Charlie by manipulating AuthReq, can disable Secure Connections and the man-in-the-middle protection flags, and, by manipulating the victims' IO capabilities, can establish a link with a LTK that is not authenticated. Similar attacks can also be performed on Bluetooth [17].

The specification comments on the security of the feature exchange (phase 1) and key establishment and optional authentication (phase 2) saying that: "phase 1 and phase 2 may be performed on a link which is either encrypted or not encrypted" [8, 2296]. However, the specification is not considering integrity protection at all, and it is not specifying how two devices should encrypt the link before pairing. In our BLE experiments, that we present in Section VII-C, *none of the devices that we test is encrypting phase 1 and 2*. As two unpaired BLE devices are not required to share any secret key, we are not surprised by this outcome. Furthermore, even if a pre-shared key is available before pairing, the encryption of phase 1 and 2 would only partially solve the problem, because if the cipher in use is malleable, then Charlie is able to downgrade the LTK's entropy anyway. What is needed in this case is *data integrity*, and the problem can be solved by using standard message authentication codes (e.g., HMAC).

B. Brute Forcing the Low Entropy BLE Encryption Key

Once Charlie is able to reduce the entropy of LTK to 7 bytes, he has to brute force it. BLE uses AES-CCM for encryption, and its entropy reduction function zeros the most significant bytes (MSB) of LTK according to the negotiated entropy value. The attacker's brute force effort for BLE is greater than Bluetooth (e.g., 7 bytes vs. 1 byte). However, we argue that 7 bytes of entropy is still insufficient for a secure connection. Prior work has sucseeded in brute forcing keys of this size by utilizing cloud computational power [47].

Additionally, we expect that (low end) BLE devices might be susceptible to low entropy boot attack [27], where a device at boot time is not capable of offering enough entropy for key generation, and ends up quietly using keys that have less entropy then expected. Considering current best practices at the time of writing, any entropy value lower than 14 bytes could be considered not secure for symmetric encryption [5].

C. KNOB Attack Root Causes for BLE

The root causes of the KNOB attack on BLE are similar to the ones of Bluetooth (see Section IV-C). The responsibility is shared between the specification of a key negotiation protocol that allows a remote attacker to downgrade the key's entropy to 7 bytes, and the lack of proper checks on the BLE implementation side. Compared to Bluetooth, BLE specifies an higher (better) minimum entropy value (7 bytes rather than 1). As for Bluetooth, we do not see a strong reason to include the possibility of downgrading the key's entropy for BLE. The entropy reduction is not improving runtime performances, and in our evaluation, presented in Section VII-C, there is no legitimate device taking advantage of this feature, e.g., proposing an entropy value lower than 16 bytes.

D. KNOB Attack Implementation for BLE

BLE's security mechanisms are implemented by the device's OS (BLE host) in a module called Security Manager (SM). By modifying the OS of a victim device we are able to simulate, among others, the effect of a remote KNOB attack without having to perform it over the air. For our implementation, we use a laptop running Linux as the victim, because we can inspect and modify Linux's Bluetooth host implementation (e.g., via bluez and the kernel) and optionally provide our own custom Bluetooth user space BLE stack.

The main advantage of the Linux stack over a custom user space one is that it offers a solid and tested implementation of the full BLE specification. In contrast, a user space stack is easier and faster to modify and use than the Linux stack (e.g., Python rather than C code, no kernel recompilations and reboots across code modifications). To perform the KNOB attack on BLE we take advantage of both functionalities by implementing the following tools:

1) *Custom linux-4.14.111 kernel.* The linux kernel contains an open source implementation of the BLE encryption key negotiation protocol (i.e., `net/bluetooth/smp.c`). We modify it such that our laptop is always proposing 7 bytes of entropy by changing line 3424 of `net/bluetooth/smp.c` to `SMP_DEV(hdev)->max_key_size = 7;` and recompiling the kernel[2].

[2]Unmodified code: https://elixir.bootlin.com/linux/v4.14.111/source/net/bluetooth/smp.c#L3424

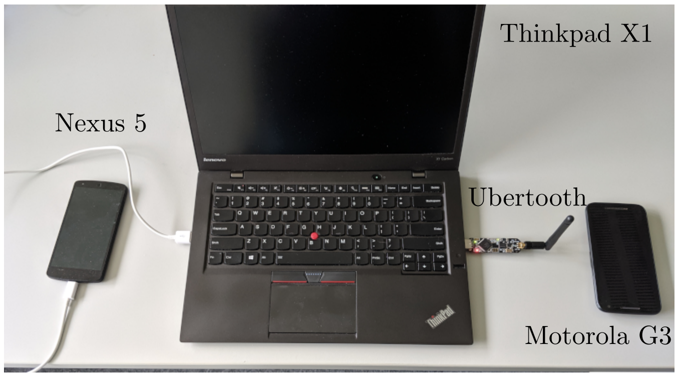

Fig. 7: Our evaluation setup. In the Bluetooth case, we use a Nexus 5 both as a victim and the attacker, a Motorola G3 as a victim, and a Thinkpad X1 laptop to patch the Nexus 5's Bluetooth firmware. In the BLE case, we use a Thinkpad X1 both as the attacker and as a victim. The laptop runs our custom Linux kernel and user space stack. In both cases, we eavesdrop the packets using an Ubertooth One.

2) *Custom user space stack*. As Linux offers the possibility to manage BLE from user space, we develop a custom BLE stack based on PyBT [39], an open-source Python package built on top of scapy [7]. PyBT provides a minimal stack, and we extend it with an API to perform adversarial manipulation of BLE pairing, including the feature exchange phase used to negotiate LTK's entropy.

VII. Evaluation

We use our implementations of the KNOB attack to successfully attack more than 17 Bluetooth chips, and 15 BLE devices. In this section we present our evaluation setup and results.

A. Evaluation Setup

Figure 7 shows the experimental setup that we use to perform our KNOB attacks on Bluetooth and BLE devices. In the Bluetooth case, we use a Nexus 5 both as a victim and the attacker, a Motorola G3 as a victim, and a Thinkpad X1 laptop to patch the Nexus 5 Bluetooth firmware. In the BLE case, we use the Thinkpad X1 both as a victim and the attacker. The laptop runs our custom Linux kernel and user space stack. In both cases, we eavesdrop the packets using an Ubertooth One [35] (firmware version 2017-03-R2). To the best of our knowledge, Ubertooth One does not capture all Bluetooth packets, but it is the only open-source, low-cost, and practical eavesdropping solution for Bluetooth, and it works well for BLE.

As our evaluation setup does not require expensive hardware, over the air manipulations, and uses open-source software, it is simple, low cost, and extendible. Each KNOB attack is easy to reproduce, and testing if a device is vulnerable is a matter of seconds. We would like to see other researchers evaluating more Bluetooth and BLE devices that currently we do not posses.

Bluetooth chip	Device(s)	Vuln?
Bluetooth version 5.0		
Snapdragon 845	Galaxy S9	✓
Snapdragon 835	Pixel 2, OnePlus 5	✓
Apple/USI 339S00428	MacBookPro 2018	✓
Apple A1865	iPhone X	✓
Snapdragon 660	Xiaomi MI A2	✓
Bluetooth version 4.2		
Intel 8265	Thinkpad X1 6th	✓
Intel 7265	Thinkpad X1 3rd	✓
Unknown	Sennheiser PXC 550	✓
Apple/USI 339S00045	iPad Pro 2	✓
BCM43438	RPi 3B, RPi 3B+	✓
BCM43602	iMac MMQA2LL/A	✓
Cambridge Silicon Unknown	Sony WH-100XM3	✓
Unknown	Bose SoundLink revolve	✓
Bluetooth version 4.1		
BCM4339 (CYW4339)	Nexus 5, iPhone 6	✓
Snapdragon 410	Motorola G3	✓
Bluetooth version $\leq$ 4.0		
Snapdragon 800	LG G2	✓
Intel Centrino 6205	Thinkpad X230	✓
Chicony Unknown	Thinkpad KT-1255	✓
Broadcom Unknown	Thinkpad 41U5008	✓
Broadcom Unknown	Anker A7721	✓
Apple W1	AirPods	*

TABLE III: List of more than 17 Bluetooth chips (24 devices) that are vulnerable to the KNOB attack. ✓indicates that a chip accepts 1 byte of entropy. * indicates that a chip accepts at least 7 bytes of entropy.

B. Bluetooth Evaluation

We conduct KNOB attacks on more than 17 Bluetooth chips from Broadcom, Qualcomm, Apple, Intel, and Chicony (by attacking 24 different devices). For each chip we conduct the KNOB attack according to Figure 7, and by following the five steps that we describe in Section V-B. For each attack we record the manipulated encryption key negotiation protocol in a pcapng file, and we manually verify the negotiation's outcome with Wireshark.

Table III shows our evaluation results. The first column contains the Bluetooth chip names. We fill the entries of this column with "Unknown" when we cannot find information about the chip manufacturer and/or model number. The second column lists the devices grouped by chip, e.g., the Snapdragon 835 is used both by the Pixel 2 and the OnePlus 5. The third column contains a ✓ if the Bluetooth chip accepts 1 byte of entropy, and a * if it accepts at least 7 bytes. We group the rows by Bluetooth version number: v5.0, v4.2, v4,1 and version lower than 4.1.

From the third column of Table III we see that all the chips accept 1 byte of entropy (✓), except the Apple W1 chip (*) that requires at least 7 bytes of entropy. Apple W1 and its successors are used in devices such as AirPods, and Apple Watches. Table III also demonstrates that the vulnerability spans across different Bluetooth versions, including the latest ones (v5.0 and v4.2). Hence, the KNOB attack is a significant threat for all Bluetooth users, and we believe that the specification has to be fixed as soon as possible.

Device	Vuln?
BLE Secure Connections	
Thinkpad X1 6rd	✓
Thinkpad X1 3rd	✓
Garmin Vivoactive 3	✓
Samsung Gear S3	✓
BLE legacy security	
Nexus 5	✓
Motorola G3	✓
Mi band	✓
Mi band 2 (x2)	✓
Fitbit Charge 2	✓
ID115 HR Plus	✓
Comet Blue thermostat	✓
Samsung TV UE48J6250	✓
EDIFIER R1280DB speaker	✓
MX Anywhere 2S	✓

TABLE IV: List of 15 BLE devices that are vulnerable to the KNOB attack. ✓ indicates that the device accepts 7 bytes of entropy.

Based on our experiments, we conclude that there are no differences between the specification and the implementation of both the Bluetooth controller (implemented in the firmware) and the Bluetooth host (implemented in the OS and usable as an interface by a Bluetooth application). In the former case, the specification is not enforcing any minimum L_{min}, and it is not protecting the entropy negotiation protocol. The firmware's implementers (to provide standard-compliant products) are allowing the negotiation of 1 byte of entropy with an insecure protocol. The only exception is the Apple W1 chip, where an attacker can reduce the entropy to 7 bytes. In the latter case, the Bluetooth specification is providing an HCI Read Encryption size API, but it is not mandating or recommending its usage, e.g., a mandatory check at the end of the LMP entropy negotiation. The host's implementers are providing this API, and the applications that we test are not using it.

C. Bluetooth Low Energy Evaluation

In Table IV we present our evaluation of the KNOB attack on 15 BLE devices. The devices are grouped in two blocks: the first block includes devices supporting BLE Secure Connections (available since Bluetooth v4.2) and the second block includes devices that are using legacy BLE security (available in Bluetooth v4.0 and 4.1. A ✓ in the second column of Table IV indicates that a device accepts 7 bytes of entropy.

As we can see from Table IV, all devices are vulnerable to the KNOB attack (i.e., in all cases we are able to downgrade LTK's entropy to 7 bytes). The tested devices include fitness bands, smart watches, laptops, and IoT devices. For each attack we record the manipulated encryption key negotiation protocol in a pcapng file, and we manually verify the negotiation's outcome with Wireshark.

Additionally, our experiments uncover a problem related to BLE's strongest security mode. In particular, BLE specifies security mode 1 with level 4 as its most secure configuration, and this mode mandates authenticated Secure Connections and a LTK with 16 bytes of entropy [8, p. 2067]. However, from our experiments we can deduce that *even if a device is using security mode 1 with level 4, the LTK's entropy can still be downgraded to 7 bytes*. For example, we are able to set LTK's entropy to 7 bytes even when the Linux kernel is compiled with security mode 1 with level 4 (i.e., its security level is set to `BT_SECURITY_FIPS`).

VIII. Countermeasures

In this section we propose legacy and non legacy compliant countermeasures to our KNOB attacks on Bluetooth and Bluetooth Low Energy.

A. Bluetooth Countermeasures

a) Legacy compliant.: Our first proposed legacy compliant countermeasure is to require a minimum and maximum amount of negotiable entropy that cannot be easily brute forced, e.g., require 16 bytes of entropy. This means fixing L_{min} and L_{max} in the Bluetooth controller (firmware), and results in the negotiation of proper encryption keys. Another possible countermeasure is to automatically have the Bluetooth host (OS) check the amount of negotiated entropy each time link layer encryption is activated, and abort the connection if the entropy does not meet a minimum requirement. The entropy value can be obtained by the host using the HCI Read Encryption Key Size Command. This solution requires to modify the Bluetooth host, and it might be suboptimal because it acts on a connection that is already established and possibly in use, and not as part of the entropy negotiation protocol. A third solution is to distrust the Bluetooth link layer, and provide the security guarantees at the application layer.

b) Non legacy compliant.: A non legacy compliant countermeasure is to secure the encryption key negotiation using the link key (K_L). The link key is a shared, and possibly authenticated, secret that should be always available before starting the entropy negotiation protocol. As the attacker should not be able to modify the victims' entropy proposals, the new encryption key negotiation protocol must provide message integrity and optional message confidentiality generating fresh keys from the link key. Preferably, the specification should get rid of the entropy negotiation protocol, and always use encryption keys with a fixed amount of entropy, e.g., 16 bytes. The implementation of these countermeasures only requires the modification of the Bluetooth controller component.

B. Bluetooth Low Energy Countermeasures

a) Legacy compliant.: Like for Bluetooth, a straightforward legacy compliant countermeasure for Bluetooth Low Energy is to require an higher minimum entropy value for the encryption key (LTK). For example, pairing can be aborted if the KeySize parameter is lower than 16 (see Figure 6). This modification requires minimal changes to the entropy negotiation's implementation (BLE host). In the case of Linux, a developer could set `SMP_MIN_ENC_KEY_SIZE = 16` in `net/bluetooth/smp.h`, and recompile the kernel. An alternative solution is to distrust the link layer and provide the security guarantees at the application layer using protocols such as BALSA [34] and TLS.

b) Non legacy compliant.: A straightforward non legacy compliant solution is to remove the possibility of negotiating the LTK's entropy while pairing, and fixing the entropy to a secure value. In particular, the KeySize parameter can be removed from the feature exchange phase, and the LTK shall always use 16 bytes of entropy. Otherwise, BLE pairing might be augmented with an initialization phase (Phase 0), where the master and the slave are establishing and authenticating a shared secret key. Then, this key can be used to protect the integrity of the feature exchange phase (including KeySize). The implementation of both countermeasures only requires the modification of the BLE host.

IX. Related Work

The most up to date survey about Bluetooth security is from NIST [36], and it recommends to use keys with 16 bytes of entropy. It also describes the key negotiation protocol, and considers it as a security issue when one of the connected devices is malicious (and not a third party). Prior surveys do not consider the problem of encryption key negotiation at all [12] or superficially discuss it [43].

The security and privacy guarantees of Bluetooth were studied since Bluetooth v1.0 [22], [46]. Particular attention was given to Secure Simple Pairing (SSP), a mechanisms that Bluetooth uses to generate and share a long term secret. Several attacks on the SSP protocol were proposed [41], [20], [17], [6]. The Key Negotiation Of Bluetooth (KNOB) attack works regardless of security guarantees provided by SSP such as mutual user authentication and the attacker is not required to observe the SSP phase.

The various implementation of Bluetooth were also analyzed and several attacks were presented on Android, iOS, Windows and Linux implementations [4]. The KNOB attack works regardless of the implementation details of the target platform, because if any implementation is standard-compliant then it is vulnerable to the KNOB attack.

The security of the ciphers used by Bluetooth was extensively discussed by cryptographers. For example, the SAFER+ cipher, used by Bluetooth for authentication purposes, was analyzed [25] and the E_0 cipher, used by Bluetooth for encryption, was also analyzed [14]. Our attack works even with perfectly secure ciphers.

For our implementation of the custom Bluetooth security procedures (presented in Section V) we used as main references the specification [8], and third-party hardware [26] and software [29] implementations.

Prior work about BLE security (Bluetooth v4.0 and v4.1) already uncovered several flaws in the design of the pairing phase. In [40] Mike Ryan demonstrated how to eavesdrop a BLE connection, and inject packets to break BLE legacy pairing. In [34], BALSA was proposed as an application layer security mechanisms alternative to BLE legacy security mechanisms. The Bluetooth v4.2 standard was updated with Secure Connection to address major weaknesses of legacy BLE security. In our work we demonstrate an attack that affects *both* legacy and Secure Connections BLE links. Application layer attacks on BLE were also demonstrated [23], our KNOB attack targets the BLE link layer, and its effects propagates to upper layers.

Detailed analysis of different BLE devices such as smart locks [19], and wearable devices [9] uncovers potential flaws in specific BLE's use cases. Our KNOB attack is generic i.e., agnostic to the specific use case. Proprietary protocols based on BLEwere also analyzed for security [1] and privacy [18] issues. Implementations of BLE were also found to be vulnerable [3]. The KNOB attack affects both proprietary protocols building on top of BLE and standard compliant implementations of the specification.

Third-party manipulations of key negotiation protocols were discussed in the context of Wi-Fi, for example key reuse in [44]. Compared to those attacks, our attack exploits not only implementation issues, but a standard-compliant vulnerability of the specification. Downgrade attacks were discussed in the context of TLS [33]. We note that in contrast to our scenario, the TLS developers have direct control over the cipher suite offered during a TLS handshake, hence they can prevent a cipher suite downgrade attack. To the best of our knowledge, this is not the case for Bluetooth, as the protocol does not enforce any mandatory checks on the (downgraded) encryption key's entropy.

X. Conclusion

In this paper we demonstrate that the entropy of any Bluetooth and Bluetooth Low Energy encryption key can be easily downgraded by a (remote) attacker to 1 byte and 7 bytes, respectively. Then, the attacker can brute force the encryption key, and break the security guarantees of affected Bluetooth or BLE link. We call our attack the Key Negotiation Of Bluetooth (KNOB) attack.

We discover the KNOB attack by finding vulnerabilities in the specifications of Bluetooth and BLE encryption key negotiation protocols. In particular, these protocols allow to insecurely negotiate low entropy values for their encryption keys. The KNOB attack is at the architectural level, hence any standard compliant Bluetooth and BLE device should be vulnerable. The attacker is not required to know any (pre) shared secrets to conduct the attack. The KNOB attack can be mounted in parallel, e.g., it can affect at the same time multiple victims even in different piconets.

We uncover the root causes of the KNOB attack, and we show how to implement the it on Bluetooth and BLE. Our implementations involve manipulations of the Bluetooth controller (implemented by the Bluetooth firmware), and the BLEhost (implemented by the OS).

To confirm that the KNOB attack is a real threat, we present the result of our large scale evaluation. We test 24 Bluetooth devices and 15 BLE devices, and all of them are vulnerable to the KNOB attack. To address the severity of the KNOB attack, we propose effective legacy and non legacy compliant countermeasures capable of neutralizing or mitigating the attack.

References

[1] Daniele Antonioli, Nils Ole Tippenhauer, and Kasper Rasmussen. Nearby Threats: Reversing, Analyzing, and Attacking Google's "Nearby Connections" on Android. In *Network and Distributed System Security Symposium (NDSS)*, February 2019.

[2] Daniele Antonioli, Nils Ole Tippenhauer, and Kasper Rasmussen. The KNOB is Broken: Exploiting Low Entropy in the Encryption Key Negotiation of Bluetooth BR/EDR. In *Proceedings of the USENIX Security Symposium*, August 2019.

[3] Armis Inc. BLEEDINGBIT Exposes Enterprise Access Points and Unmanaged Devices to Undetectable Chip Level Attack. https://armis.com/bleedingbit/, Accessed: 2019-07-24.

[4] Armis Inc. The Attack Vector BlueBorne Exposes Almost Every Connected Device. https://armis.com/blueborne/, Accessed: 2018-01-26.

[5] Elaine Barker, William Barker, William Burr, William Polk, and Miles Smid. Recommendation for key management part 1: General (revision 3). *NIST special publication*, 800(57):1–147, 2012.

[6] Eli Biham and Lior Neumann. Breaking the bluetooth pairing–fixed coordinate invalid curve attack. http://www.cs.technion.ac.il/~biham/BT/bt-fixed-coordinate-invalid-curve-attack.pdf, Accessed: 2018-10-30.

[7] Philippe Biondi. Scapy: Packet crafting for python2 and python3. https://scapy.net/, Accessed: 2018-01-26.

[8] Bluetooth SIG. Bluetooth Core Specification v5.0. https://www.bluetooth.org/DocMan/handlers/DownloadDoc.ashx?doc_id=421043, Accessed: 2018-10-28, 2016.

[9] Jiska Classen, Daniel Wegemer, Paul Patras, Tom Spink, and Matthias Hollick. Anatomy of a vulnerable fitness tracking system: Dissecting the fitbit cloud, app, and firmware. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies*, 2018.

[10] Bob Cromwell. The Problem With Government-Imposed Backdoors. https://cromwell-intl.com/cybersecurity/backdoors.html, Accessed: 2019-2-4.

[11] Arnaud Delmas. A C implementation of the Bluetooth stream cipher E0. https://github.com/adelmas/e0, Accessed: 2018-10-28.

[12] John Dunning. Taming the blue beast: A survey of bluetooth based threats. *IEEE Security & Privacy*, 8(2):20–27, 2010.

[13] Kassem Fawaz, Kyu-Han Kim, and Kang G Shin. Protecting privacy of {BLE} device users. In *25th {USENIX} Security Symposium ({USENIX} Security 16)*, pages 1205–1221, 2016.

[14] Scott Fluhrer and Stefan Lucks. Analysis of the E0 encryption system. In *International Workshop on Selected Areas in Cryptography*, pages 38–48. Springer, 2001.

[15] Glenn Greenwald. *No place to hide: Edward Snowden, the NSA, and the US surveillance state*. Metropolitan Books, 2014.

[16] Kent Griffin, John Hastings Granbery, Hill Ferguson, David Marcus, and Michael Charles Todasco. Bluetooth low energy (ble) pre-check in, 2015. US Patent App. 14/479,200.

[17] Keijo Haataja and Pekka Toivanen. Two practical man-in-the-middle attacks on bluetooth secure simple pairing and countermeasures. *IEEE Transactions on Wireless Communications*, 9(1), 2010.

[18] Hexway. Apple bleee. Everyone knows What Happens on Your iPhone . https://hexway.io/blog/apple-bleee/, Accessed: 2019-07-24.

[19] Grant Ho, Derek Leung, Pratyush Mishra, Ashkan Hosseini, Dawn Song, and David Wagner. Smart locks: Lessons for securing commodity internet of things devices. In *Proceedings of the 11th ACM on Asia conference on computer and communications security*, pages 461–472. ACM, 2016.

[20] Konstantin Hypponen and Keijo MJ Haataja. "nino" man-in-the-middle attack on bluetooth secure simple pairing. In *2007 3rd IEEE/IFIP International Conference in Central Asia on Internet*, pages 1–5. IEEE, 2007.

[21] IETF. Counter with CBC-MAC (CCM). https://www.ietf.org/rfc/rfc3610.txt, Accessed: 2018-10-28.

[22] Markus Jakobsson and Susanne Wetzel. Security weaknesses in Bluetooth. In *Cryptographers' Track at the RSA Conference*, pages 176–191. Springer, 2001.

[23] Sławomir Jasek. Gattacking bluetooth smart devices. In *Black Hat USA Conference*, 2016.

[24] Avinash Kak. BitVector.py. https://engineering.purdue.edu/kak/dist/BitVector-3.4.8.html, Accessed: 2018-10-28.

[25] John Kelsey, Bruce Schneier, and David Wagner. Key schedule weaknesses in SAFER+. In *The Second Advanced Encryption Standard Candidate Conference*, pages 155–167, 1999.

[26] Paraskevas Kitsos, Nicolas Sklavos, Kyriakos Papadomanolakis, and Odysseas Koufopavlou. Hardware implementation of Bluetooth security. *IEEE Pervasive Computing*, (1):21–29, 2003.

[27] Sandeep Kumar, Christof Paar, Jan Pelzl, Gerd Pfeiffer, and Manfred Schimmler. Breaking ciphers with copacobana–a cost-optimized parallel code breaker. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 101–118. Springer, 2006.

[28] Jiun-Ren Lin, Timothy Talty, and Ozan K Tonguz. On the potential of bluetooth low energy technology for vehicular applications. *IEEE Communications Magazine*, 53(1):267–275, 2015.

[29] Musaria K Mahmood, Lujain S Abdulla, Ahmed H Mohsin, and Hamza A Abdullah. MATLAB Implementation of 128-key length SAFER+ Cipher System.

[30] Dennis Mantz, Jiska Classen, Matthias Schulz, and Matthias Hollick. Internalblue - bluetooth binary patching and experimentation framework. In *Proceedings of Conference on Mobile Systems, Applications and Services (MobiSys)*. ACM, June 2019.

[31] James L Massey, Gurgen H Khachatrian, and Melsik K Kuregian. Nomination of SAFER+ as candidate algorithm for the Advanced Encryption Standard (AES). *NIST AES Proposal*, 1998.

[32] Yan Michalevsky, Suman Nath, and Jie Liu. Mashable: mobile applications of secret handshakes over bluetooth le. In *Proceedings of the 22nd Annual International Conference on Mobile Computing and Networking*, pages 387–400. ACM, 2016.

[33] Bodo Möller, Thai Duong, and Krzysztof Kotowicz. This POODLE bites: exploiting the SSL 3.0 fallback. https://www.openssl.org/~bodo/ssl-poodle.pdf, Accessed: 2019-02-04, 2014.

[34] Diego A Ortiz-Yepes. Balsa: Bluetooth low energy application layer security add-on. In *2015 International Workshop on Secure Internet of Things (SIoT)*, pages 15–24. IEEE, 2015.

[35] Michael Ossmann. Project Ubertooth. https://github.com/greatscottgadgets/ubertooth, Accessed: 2018-11-01.

[36] John Padgette. Guide to bluetooth security. *NIST Special Publication*, 800:121, 2017.

[37] Christina Pöpper, Nils Ole Tippenhauer, Boris Danev, and Srdjan Čapkun. Investigation of signal and message manipulations on the wireless channel. In *Proceedings of the European Symposium on Research in Computer Security (ESORICS)*, December 2011.

[38] Jordan Robertson and Michael Riley. The Big Hack: How China Used a Tiny Chip to Infiltrate U.S. Companies. https://www.bloomberg.com/news/features/2018-10-04/the-big-hack-how-china-used-a-tiny-chip-to-infiltrate-america-s-top-companies, Accessed: 2018-10-30.

[39] Mike Ryan. Pybt: Hackable bluetooth stack in python. https://github.com/mikeryan/PyBT, Accessed: 2019-06-19.

[40] Mike Ryan. Bluetooth: With low energy comes low security. In *Proceedings of USENIX Workshop on Offensive Technologies (WOOT)*, volume 13, pages 4–4, 2013.

[41] Yaniv Shaked and Avishai Wool. Cracking the Bluetooth PIN. In *Proceedings of the conference on Mobile systems, applications, and services (MobiSys)*, pages 39–50. ACM, 2005.

[42] Google Cloud Team. Google titan security keys. https://cloud.google.com/titan-security-key/, Accessed: 2019-02-04, 2018.

[43] Juha T Vainio. Bluetooth security. In *Proceedings of Helsinki University of Technology, Telecommunications Software and Multimedia Laboratory, Seminar on Internetworking: Ad Hoc Networking, Spring*, volume 5, 2000.

[44] Mathy Vanhoef and Frank Piessens. Key reinstallation attacks: Forcing nonce reuse in WPA2. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 1313–1328. ACM, 2017.

[45] Matthias Wilhelm, Ivan Martinovic, Jens B Schmitt, and Vincent Lenders. Short paper: reactive jamming in wireless networks: how realistic is the threat? In *Proceedings of the fourth ACM conference on Wireless network security*, pages 47–52. ACM, 2011.

[46] Ford-Long Wong and Frank Stajano. Location privacy in Bluetooth. In *European Workshop on Security in Ad-hoc and Sensor Networks*, pages 176–188. Springer, 2005.

[47] JunWeon Yoon, TaeYoung Hong, JangWon Choi, ChanYeol Park, KiBong Kim, and HeonChang Yu. *Evaluation of P2P and cloud computing as platform for exhaustive key search on block ciphers*. Springer US, 2018.

[48] Bin Yu, Lisheng Xu, and Yongxu Li. Bluetooth low energy (ble) based mobile electrocardiogram monitoring system. In *2012 IEEE International Conference on Information and Automation*, pages 763–767. IEEE, 2012.