The KNOB is Broken: Exploiting Low Entropy in the Encryption Key Negotiation Of Bluetooth BR/EDR

Daniele Antonioli
Singapore University of Technology and Design
daniele_antonioli@mymail.sutd.edu.sg

Nils Ole Tippenhauer
CISPA Helmholtz Center for Information Security
tippenhauer@cispa.saarland

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

Abstract

We present an attack on the encryption key negotiation protocol of Bluetooth BR/EDR. The attack allows a third party, without knowledge of any secret material (such as link and encryption keys), to make two (or more) victims agree on an encryption key with only 1 byte (8 bits) of entropy. 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). The attack is stealthy because the encryption key negotiation is transparent to the Bluetooth users. The attack is standard-compliant because all Bluetooth BR/EDR versions require to support encryption keys with entropy between 1 and 16 bytes and do not secure the key negotiation protocol. As a result, *the attacker completely breaks Bluetooth BR/EDR security without being detected*. We call our attack **Key Negotiation Of Bluetooth (KNOB)** attack.

The attack targets the firmware of the Bluetooth chip because the firmware (Bluetooth controller) implements all the security features of Bluetooth BR/EDR. As a standard-compliant attack, it is expected to be effective on any firmware that follows the specification and on any device using a vulnerable firmware. We describe how to perform the KNOB attack, and we implement it. We evaluate our implementation on more than 14 Bluetooth chips from popular manufacturers such as Intel, Broadcom, Apple, and Qualcomm. Our results demonstrate that all tested devices are vulnerable to the KNOB attack. We discuss countermeasures to fix the Bluetooth specification and its implementation.

1 Introduction

Bluetooth BR/EDR (referred for the rest of this paper as Bluetooth), is a short-range wireless technology widely used by many products such as mobile devices, laptops, IoT and industrial devices. Bluetooth provides security mechanisms to achieve authentication, confidentiality and data integrity at the link layer [6, p. 1646].

The security and privacy of Bluetooth has been attacked and fixed several times, going all the way back to Bluetooth v1.0. [15, 32]. Several successful attacks on the (secure simple) pairing phase [28, 13, 4] have resulted in substantial revisions of the standard. Attacks on Android, iOS, Windows and Linux implementations of Bluetooth were also discussed in [2]. However, little attention has been given to the security of the *encryption key negotiation protocol*, e.g., the Bluetooth security overview in the latest Bluetooth core specification (v5.0) does not mention it [6, p. 240].

The encryption key negotiation protocol is used by two Bluetooth devices to agree on the entropy of the link layer encryption key. Entropy negotiation was introduced in the specification of Bluetooth to cope with international encryption regulations and to facilitate security upgrades [6, p. 1650]. To the best of our knowledge, all versions of the Bluetooth standard (including the latest v5.0 [6]) *require* to use entropy values between 1 and 16 bytes. The specification of Bluetooth states this requirement as follows: "For the encryption algorithm, the key size may vary between 1 and 16 octets (8 - 128 bits)" [6, p. 1650]. Our interpretation of this requirement is that any device to be standard-compliant has to support encryption keys with entropy varying from one to sixteen bytes. The attack that we present in this work confirms our interpretation.

The encryption key negotiation protocol is conducted between two parties as follows: the initiator proposes an entropy value N that is an integer between 1 and 16, the other party either accepts it or proposes a lower value or aborts the protocol. If the other party proposes a lower value, e.g., $N-1$, then the initiator either accepts it or proposes a lower value or it aborts the protocol. At the end of a successful negotiation the two parties have agreed on the entropy value of the Bluetooth encryption key. The entropy negotiation is performed over the Link Manager Protocol (LMP), it is not encrypted and not authenticated, and it is transparent to the Bluetooth users because LMP packets are managed by the firmware of the Bluetooth chips and they are not propagated to higher layers [6, p. 508].

In this paper we describe, implement and evaluate an attack capable of making two (or more) victims using a Bluetooth encryption key with 1 byte of entropy without noticing it. The attacker then can easily brute force the encryption key, eavesdrop and decrypt the ciphertext and inject valid ciphertext without affecting the status of the target Bluetooth piconet. In other words, *the attacker completely breaks Bluetooth BR/EDR security without being detected*. We call this attack the **Key Negotiation Of Bluetooth (KNOB)** attack.

The KNOB attack can be conducted remotely or by maliciously modifying few bytes in one of the victim's Bluetooth firmware. Being a standard-compliant attack it is expected to be effective on any firmware implementing the Bluetooth specification, regardless of the Bluetooth version. The attacker is not required to posses any (pre-shared) secret material and he does not have to observe the pairing process of the victims. The attack is effective even when the victims use the strongest security mode of Bluetooth (Secure Connections). The attack is stealthy because the application using Bluetooth and even the operating systems of the victims cannot access or control the encryption key negotiation protocol (see Section 3.2 for the details).

After explaining the attack in detail, we implement it leveraging our development of several Bluetooth security procedures to generate valid link and encryption keys, and the InternalBlue toolkit [21]. Our implementation allows a man-in-the-middle attacker to intercept, manipulate, and drop LMP packets in real-time and to brute force low-entropy encryption keys, without knowing any (pre-shared) secret. We have disclosed our findings about the KNOB attack with CERT and the Bluetooth SIG, and following that, we plan to release our tools as open-source at `https://github.com/francozappa/knob`. This will enable other Bluetooth researchers to take advantage of our work.

We summarize our main contributions as follows:

- We develop an attack on the encryption key negotiation protocol of Bluetooth BR/EDR that allows to let two unaware victims negotiate a link-layer encryption key with 1 byte of entropy. The attacker then is able to brute force the low entropy key, decrypt all traffic and inject arbitrary ciphertext. The attacker does not have to know any secret material and he can target multiple nodes and piconets at the same time.

- We demonstrate the practical feasibility of the attack by implementing it. Our implementation involves a man-in-the-middle attacker capable of manipulating the encryption key negotiation protocol, brute forcing the key and decrypting the traffic exchanged by two (or more) unaware victims.

- All standard-compliant devices should be vulnerable to our attack, including the ones using the strongest Bluetooth security mode. In order to demonstrate that this problem has not somehow been fixed in practice, we test more than 14 different Bluetooth chips and find all of them to be vulnerable.

- We discuss what changes should be made, both to the Bluetooth standard and its implementation, in order to counter this attack.

Our work is organized as follows: in Section 2 we introduce the Bluetooth BR/EDR stack. In Section 3 we present the Key Negotiation Of Bluetooth (KNOB) attack. An implementation of the attack is discussed in Section 4. We evaluate the impact of our attack in Section 5 and we discuss the attack and our proposed countermeasures in Section 6. We present the related work in Section 7. We conclude the paper in Section 8.

2 Background

2.1 Bluetooth Basic Rate/Extended Data Rate

Bluetooth Basic Rate/Extended Data Rate (BR/EDR), also known as Bluetooth Classic, is a widely used wireless technology for low-power short-range communications maintained by the Bluetooth Special Interest Group(SIG) [6]. Its physical layer uses the same 2.4 GHz frequency spectrum of WiFi and (adaptive) frequency hopping to mitigate RF interference. A Bluetooth network is called a piconet and it uses a master-slave medium access protocol. There is always one master device per piconet at a time. The devices are synchronized by maintaining a reference clock signal, defined as CLK. Each device has a Bluetooth address (BTADD) consisting of a sequence of six bytes. From left to right, the first two bytes 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*, indicated with K_L. There are four types of link key: initialization, unit, combination and master. A initialization key is always generated for each new pairing procedure. A unit key is generated from a device and utilized to pair with every other device, and its usage is not recommended because it is insecure. A combination key is generated using Elliptic Curve Diffie Hellman (ECDH) on the P-256 elliptic curve. This procedure is defined as Secure Simple Pairing (SSP) and it provides optional authentication of the link key. Combination keys are the most secure and widely used. A master key is generated only for broadcast encryption and it has limited usage. The master key is temporary, while the others are semi-permanent. A semi-permanent key can persist until a new link key is requested (link key is bonded) or it can change within the same session (link key is not bonded). In this paper we deal with combination link keys generated using authenticated SSP.

The specification of Bluetooth 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 two values: the Signed Response (SRES) and the 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) parameter, together with K_L and a public nonce (EN_RAND). E_1 and E_3 use a custom hash function defined in the specification of Bluetooth with H. The hash function is based on SAFER+, a block cipher that was submitted as an AES candidate in 1998 [22].

Once the encryption key K_C is generated there are two possible 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 [14]. As a side note, the specification of Bluetooth defines a message authentication codes (MAC) with the term message integrity check (MIC). If Secure Connections is not supported then the devices use the E_0 stream cipher for encryption. The cipher is derived from the Massey-Rueppel algorithm and it is described in the specification of Bluetooth [6, p. 1662]. E_0 requires synchronization between the master and the slaves of the piconet, this is achieved using the Bluetooth's clock value (CLK).

Modern implementations of Bluetooth provides the Host Controller Interface (HCI). This interface allows to separate the Bluetooth 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 Bluetooth chip 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 Link Manager Protocol (LMP) is used over the air by two controllers to perform link set-up and control for Bluetooth BR/EDR. LMP is neither encrypted nor authenticated. The LMP packets do not propagate to higher protocol layers, hence, the hosts (OSes) are not aware about the LMP packets exchanged between the Bluetooth controllers.

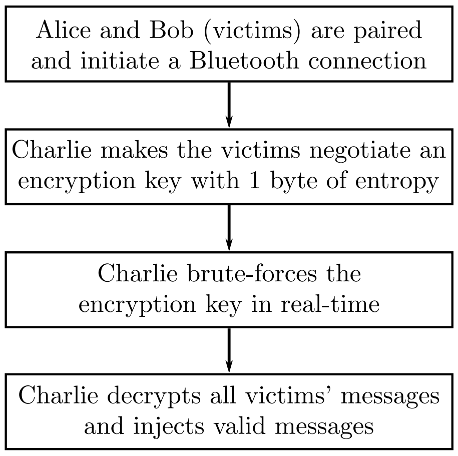
Alice and Bob (victims) are paired and initiate a Bluetooth connection
Charlie makes the victims negotiate an encryption key with 1 byte of entropy
Charlie brute-forces the encryption key in real-time
Charlie decrypts all victims' messages and injects valid messages

Figure 1: High level stages of a KNOB attack.

3 Exploiting Low Entropy in the Encryption Key Negotiation Of Bluetooth BR/EDR

In this section we describe the Key Negotiation Of Bluetooth (KNOB) attack. The attack allows Charlie (the attacker) to reduce the entropy of the encryption key of any Bluetooth BR/EDR (referred as Bluetooth) connection to 1 byte, without being detected by the victims (Alice and Bob). The attacker can brute force the encryption key without having to know any (pre-shared) secret material and without having to observe the Secure Simple Pairing protocol. As a result, the attacker can eavesdrop and decrypt all the traffic and inject arbitrary packets in the target Bluetooth network (piconet). The attack works regardless the usage of Secure Connections (the strongest security mode of Bluetooth). The KNOB attack high level stages are shown in Figure 1 and they are described in detail in the rest of this section.

3.1 System and Attacker Model

We assume a system composed of two or more legitimate devices that communicate using Bluetooth (as described in Section 2). One device is the master and the others are slaves. Without loss of generality, we focus on a piconet with one master and one slave (Alice and Bob). We indicate their Bluetooth addresses with BTADD$_M$ and BTADD$_S$, and the Bluetooth clock with CLK. The clock is used for synchronization and it does not provide any security guarantee. 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. For example, if both devices have a display their users have to confirm that they see the same numeric sequence to mutually authenticate.

The attacker (Charlie) wants to decrypt all messages exchanged between Alice and Bob and inject valid encrypted messages, without being detected. The attacker has no access

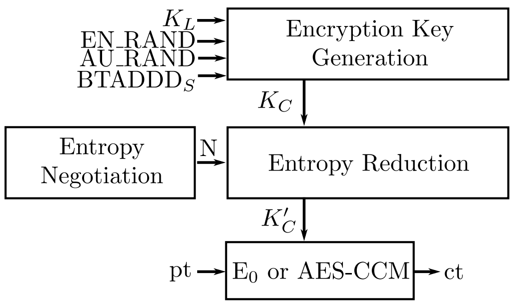

Figure 2: Generation and usage of the Bluetooth link layer encryption key (K'_C). Firstly, K_C is generated from K_L and other public parameters. K_C has 16 bytes of entropy, and it is not directly used as the encryption key. K'_C, the actual encryption key, is computed by reducing the entropy of K_C to N bytes. N is an integer between 1 and 16 and it is the result of the encryption key negotiation protocol. The N byte entropy K'_C is then used for link layer encryption by either the E_0 or the AES-CCM cipher.

to any (pre-shared) secret material. i.e., the link key K_L and the encryption key K_C. Charlie can observe the public nonces (EN_RAND and AU_RAND), the Bluetooth clock and the packets exchanged between Alice and Bob.

We define two attacker models: a *remote* attacker and a *firmware* attacker. A remote attacker controls a device that is in Bluetooth range with Alice and Bob. He is able to passively capture 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 [31, 26]. The firmware attacker is able to compromise the firmware of the Bluetooth chip of a single victim using techniques such as backdoors [7], supply-chain implants [12], and rogue chip manufacturers [27]. The firmware attacker requires no access to the Bluetooth host (OS) and applications used by the victims.

3.2 Negotiate a Low Entropy Encryption Key

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 2). 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.

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. N is the outcome of the Bluetooth *encryption key negotiation protocol* (Entropy Negotiation in Figure 2). The protocol is part of the Bluetooth specification since version v1.0, and it was introduced to cope with international encryption regulations and

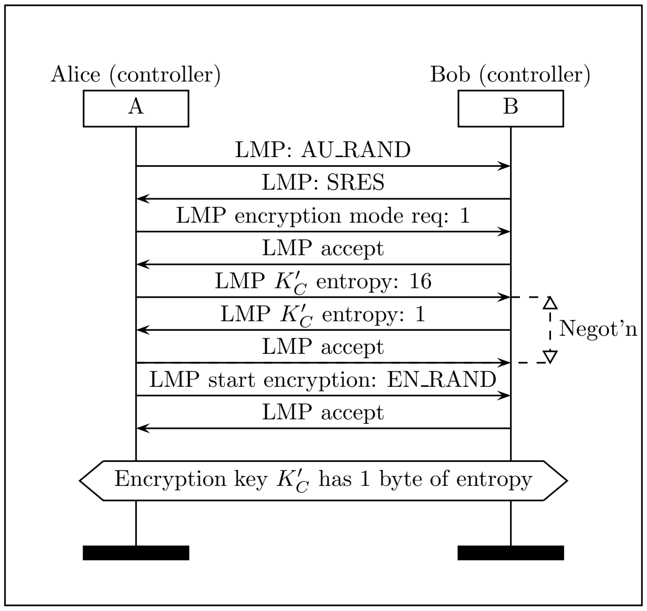

Figure 3: Alice and Bob negotiate 1 byte of entropy for the encryption key (K'_C). The protocol is run by Alice and Bob controllers (implemented in their Bluetooth chip) over the air using LMP.

to facilitate security upgrades [6, p. 1650]. The specification of the Bluetooth encryption key negotiation protocol contains three significant problems:

1. It allows to negotiate entropy values as low as 1 byte, regardless the Bluetooth security level.
2. It is neither encrypted nor authenticated.
3. It is implemented in the Bluetooth controller (firmware) and it is transparent to the Bluetooth host (OS) and to the user of a Bluetooth application.

Hence, an attacker (Charlie) can convince any two victims (Alice and Bob) to negotiate N equal to 1, the lowest possible, yet standard-compliant, entropy value. As a result the victims compute and use a Bluetooth encryption key (K'_C) with one byte of entropy. The victims (and their OSes) are not aware about the entropy reduction of K'_C because the negotiation happens between the victims' Bluetooth controller (firmware) and the packets do not propagate to the victims' Bluetooth host (OS).

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. In the following, we provide an example where Alice (the master) proposes 16 bytes of entropy, and Bob (the slave) is only able to support 1 byte of entropy (see Figure 3). The standard enables to set the minimum and maximum entropy

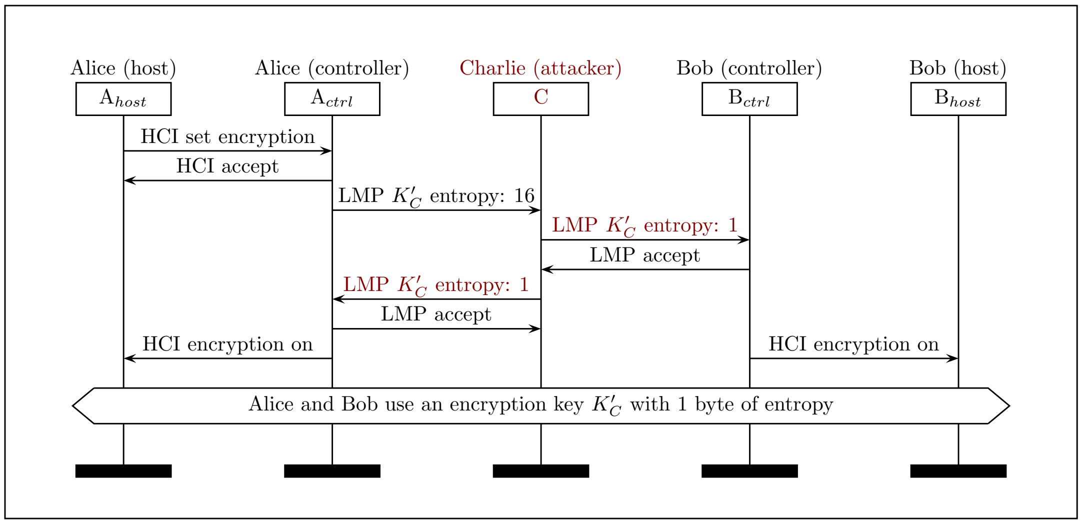

Figure 4: The KNOB attack sets the entropy of the encryption key (K'_C) to 1 byte. Alice requests Bob to activate encryption and starts the encryption key negotiation protocol. The attacker (Charlie) changes the entropy suggested by Alice from 16 to 1 byte. Bob accepts Alice's proposal and Charlie changes Bob's acceptance to a proposal of 1 byte. Alice, who originally proposed 16 bytes of entropy and she is asked to use 1 byte accepts the (standard-compliant) proposal. Charlie drops Alice's acceptance message because Bob already accepted Alice's proposal (modified by Charlie). Charlie does not know any pre-shared secret and does not observe SSP.

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 Alice's Bluetooth firmware declares $L_{max} = 16$ and $L_{min} = 1$, and Bob's Bluetooth firmware declares $L_{max} = L_{min} = 1$.

The encryption key negotiation protocol is carried over the Link Manager Protocol (LMP). The first two messages in Figure 3 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 3). Alice proposes 16 bytes of entropy. Bob can either propose a smaller value or accept the proposed one or abort the negotiation. In our example, Bob proposes 1 byte of entropy because it is the only value that he supports and Alice accepts it. Then, Alice requests to activate link-layer encryption and Bob accepts. Finally, Alice and Bob compute the same encryption key (K'_C) that has 1 byte of entropy. Note that, the Bluetooth hosts of Alice and Bob do not have access to K_C and K'_C, they are only informed about the outcome of the negotiation. The key negotiation procedure can also be initiated by the Bob (the slave), resulting in the same outcome.

Figure 4 describes how the attacker (Charlie) manages to let Alice and Bob agree on a K'_C with 1 byte of entropy when both Alice and Bob declare $L_{max} = 16$ and $L_{min} = 1$. In this Figure we also show the local interactions between hosts and controllers to emphasize that at the end of the negotiation the hosts are not informed about N and K'_C.

The attack is performed as follows: Alice's Bluetooth host requests to activate (set) encryption. Alice's Bluetooth controller accepts the local requests and starts the encryption key negotiation procedure with Bob's Bluetooth controller over the air. The attacker intercepts Alice's proposed key entropy and substitutes 16 with 1. This simple substitution works because LMP is neither encrypted nor integrity protected. Bob's controller accepts 1 byte. The attacker intercepts Bob's acceptance message and change it to an entropy proposal of 1 byte. Alice thinks that Bob does not support 16 bytes of entropy and accepts 1 byte. The attacker intercepts Alice' acceptance message and drops it. Finally, the controllers of Alice and Bob compute the same K'_C with one byte of entropy and notify their respective hosts that link-layer encryption is on.

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." [6, 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 5, 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 participating in the entropy negotiation protocol. 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 low entropy negotiation presented in Figure 4 can be performed by both attacker models presented in Section 3.1. The remote attacker has the capabilities of dropping and injecting valid plaintext (the encryption key negotiation protocol is neither encrypted nor authenticated). The firmware attacker can modify few bytes in the Bluetooth firmware of a victim to always negotiate 1 byte of entropy. Furthermore, the negotiation is effective regardless of who initiates the protocol and the roles (master or slave) of the victims in the piconet.

3.3 Brute forcing the 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. 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 valid plaintext contains well known Bluetooth 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 2). The specification of Bluetooth calls this procedure Encryption Key Size Reduction [6].

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

In case of E_0, K'_C is computed using Equation (E_s), where N is an integer between 1 and 16 resulted from the encryption key negotiation protocol (see Section 3.2). $g_1^{(N)}$ 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 these polynomials depend on N and they are tabulated in [6, p. 1668]. If $N=1$, then we can 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. Table 4 in Appendix A shows the first twenty K'_C with 1 byte of entropy for E_0 and AES-CCM. More details about the brute force implementation are discussed in Section 4.

3.4 KNOB Attack Implications

The Key Negotiation Of Bluetooth (KNOB) attack exploits a vulnerability at the architectural level of Bluetooth. The vulnerable encryption key negotiation protocol endangers potentially all standard compliant Bluetooth devices, regardless their Bluetooth version number and implementation details. We believe that the encryption key negotiation protocol has to be fixed as soon as possible.

In particular the KNOB attack has serious implications related to its *effectiveness*, *stealthiness*, and *cost*. The attack is effective because it exploits a weakness in the specification of Bluetooth. The Bluetooth security mode does not matter, i.e., the attack works even with Secure Connections. The implementation details do not matter, e.g., whether Bluetooth is implemented in hardware or in software. The time constraints imposed by the Bluetooth protocols do not matter because the attacker can eavesdrop the traffic and brute force the low-entropy key offline. The type of connection does not matter, e.g., the attack works with long-lived and short-lived connections. In a long-lived connection, e.g., victims are a laptop and a Bluetooth keyboard, the attacker has to negotiate and brute force a single low-entropy K'_C. In a short-lived connection, e.g., victims are two devices transferring files over Bluetooth, the attacker has to negotiate and brute force multiple low-entropy K'_C over time re-using the same attack technique without incurring in significant runtime and computational overheads.

The attack is stealthy because only the Bluetooth controllers (implemented in the victims' Bluetooth chip) are aware of N and K'_C. By design, the controllers are not notifying the Bluetooth hosts (implemented in the OSes) about N, but only about the outcome of the entropy negotiation. The users and the Bluetooth application developers are unaware of this problem because they use Bluetooth link-layer encryption as a trusted service.

The attack is cheap because it does not require a strong attacker model and expensive resources to be conducted. We expect that a remote attacker with commercial-off-the-shelf devices such as a software defined radio, GNU Radio and a laptop can conduct the attack.

3.5 KNOB Attack Root Causes

The root causes of the KNOB attack are shared between the specification and the implementation of Bluetooth BR/EDR 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 5), the Bluetooth applications that we tested are failing to check the negotiated entropy in practice. This is understandable because they are implementing a specification that is not mandating or explicitly 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 5) we observe that if two devices are not attacked they always use it in the same way (a device proposes 16 bytes of entropy and the other accepts). Furthermore, the entropy reduction does not improve runtime performances because the size of the encryption key is fixed to 16 bytes even when its entropy is reduced.

4 Implementation

We now discuss how we implemented the KNOB attack using a reference attack scenario. In particular, 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 a legitimate device (as in Figure 2). In our attack scenario, 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 of an application layer protocol in the TCP/IP stack.

Our implementation required significant efforts mainly due to the lack of low-cost Bluetooth protocol analyzers and software libraries implementing the custom Bluetooth security primitives (such as the modified SAFER+ block cipher). Using our implementation we conducted successful KNOB attacks on more than 14 different Bluetooth chips, the attacks are evaluated in Section 5.

4.1 Attack Scenario

To describe our implementation we use an attack scenario with two victims a Nexus 5 and a Motorola G3, Table 1 lists their relevant specifications. The Nexus 5 is used also as a man-in-the-middle attacker by adding extra code to its Bluetooth firmware. This setup allows us to *simulate* a remote man-in-the-middle attacker (more details in Section 4.2). To perform eavesdropping, we use an Ubertooth One [24] with firmware version 2017-03-R2 (API:1.02). To the best of our knowledge, Ubertooth One does not capture all Bluetooth BR/EDR packets, but it is the only open-source, low-cost, and practical eavesdropping solution for Bluetooth that we know about. To brute force K'_C and decrypt the ciphertext we use a ThinkPad X1 laptop running a Linux based OS.

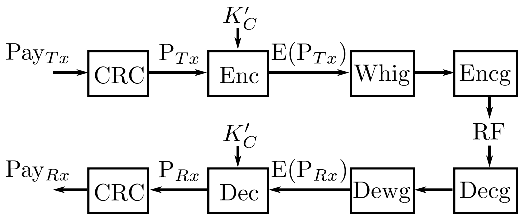
$$\begin{tikzcd}
\text{Pay}_{Tx} \arrow[r] & \boxed{\text{CRC}} \arrow[r, "\text{P}_{Tx}"] & \boxed{\text{Enc}} \arrow[r, "\text{E}(\text{P}_{Tx})"] & \boxed{\text{Whig}} \arrow[r] & \boxed{\text{Encg}} \arrow[d] \\
 & & K'_C \arrow[u] & & \text{RF} \arrow[d] \\
\text{Pay}_{Rx} & \boxed{\text{CRC}} \arrow[l] & \boxed{\text{Dec}} \arrow[l, "\text{P}_{Rx}"'] & \boxed{\text{Dewg}} \arrow[l, "\text{E}(\text{P}_{Rx})"'] & \boxed{\text{Decg}} \arrow[l] \\
 & & K'_C \arrow[u] & &
\end{tikzcd}$$

Figure 5: 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.

The victims use the following security procedures: 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 works also with Secure Connections.

Every E_0-encrypted packet that contains data is transmitted and received as in Figure 5. 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. 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. The encryption and decryption procedures are the same because E_0 is a stream cipher, i.e., the same keystream is XORed with the plaintext and the ciphertext. Whitening and encoding procedures do not add any security guarantee and the Ubertooth One is capable of performing both procedures.

4.2 Manipulation of the Entropy Negotiation

We implement the manipulation of the encryption key negotiation protocol (presented in Section 3.2) by extending the functionalities of InternalBlue [21] and using it to patch the Bluetooth chip 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 Nexus 5 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) as a victim and a remote KNOB attacker without having to deal with the practical issues related with wireless attacks over-the-air.

InternalBlue is an open-source toolkit capable of interfacing with the firmware of the BCM4339 Bluetooth chip in

		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 1: 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.

Nexus 5 phones. To use it, one has to root the target Nexus 5 and compile and install the Android Bluetooth stack with debugging features enabled. InternalBlue allows to patch the firmware in real-time (e.g., start LMP monitoring) and read the ROM and the RAM of firmware at runtime. InternalBlue provides a way to hook and execute arbitrary code in the Bluetooth firmware. At the time of writing, InternalBlue is not capable of hooking directly the key negotiation logic. However, we managed to extend it to enable two victims (one is always the Nexus 5) to negotiate one (or more) byte of entropy.

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 the Secure Simple Pairing process. 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 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 4.
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 extended InternalBlue to perform the fourth step of the list. In this context, the most important file of InternalBlue is `internalblue/fw_5.py`. This file contains all the information about the 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 upgraded them to be used also for LMP manipulation.

Listing 1 shows three ARM assembly code blocks that we added to `fw_5.py` to let the Nexus 5 and the Motorola G3 negotiate 1 byte of entropy. In this case 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.

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 (assuming that she is vulnerable). 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 developed and used similar patches to cover the other 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.

4.3 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 the key value (key space is 256). In this section we explain how we brute forced and validated a E_0 encryption key with 1 byte of entropy. The key was used in one of our KNOB attacks to decrypt the content of a file transferred over a link layer encrypted Bluetooth connection.

The details about the E_0 encryption scheme are presented in Figure 6, 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

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:
```

is a public parameter. We did not have to implement the E_0 cipher because we found an open-source implementation [8] which we verified against the specification of Bluetooth. To provide valid K'_C candidates to E_0 we had to implement the E_s entropy reduction procedure. This procedure 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 [16] Python module to perform such computations.

Our Python brute force script takes a ciphertext (captured over the air using Ubertooth One) and tries to decrypt it by using the E_0 cipher with all possible values of K'_C. We validate our script by decrypting the content of a file sent from the Nexus 5 to the Motorola G3 using the OBEX Bluetooth profile after the negotiation of 1 byte of entropy. The content of the file (in ASCII) is `aaaabbbbccccdddd`. We discuss several brute forcing practical issues in Section 6.3.

Once we found the matching plaintext we wanted to verify that the brute forced key was effectively the one in use by the victims. To do that we had to implement E_1 and E_3, the former

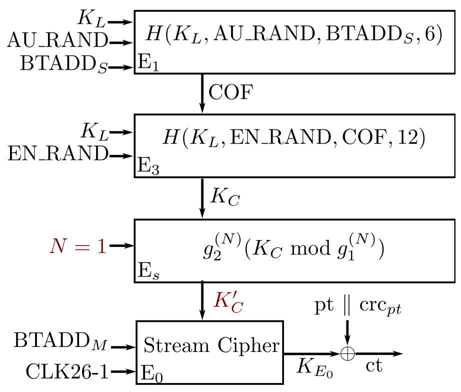

Figure 6: Implementation of the KNOB attack on the E_0 cipher. The attacker makes the victims agree on a K'_C with one byte of entropy ($N = 1$) and then brute force K'_C, without knowing K_L and K_C.

is used to compute the Ciphering Offset Number (COF), the latter to compute K_C (see Figure 6). Both procedures use a custom hash function defined in the specification of Bluetooth with H. We write E_1 and E_3 equations and label them with their respective names as follows:

$$SRES\|ACO = H(K_L, \text{AU\_RAND}, \text{BTADD}_S, 6) \qquad (E_1)$$
$$K_C = H(K_L, \text{EN\_RAND}, \text{COF}, 12) \qquad (E_3)$$

Figure 7 shows how E_3 uses the H hash function, H internally uses SAFER+, a block cipher that was submitted as an AES candidate in 1998 [22]. SAFER+ is used with 128 bit block size (8 rounds), in ECB mode, and only for encryption. SAFER+' (SAFER+ prime) 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 of Bluetooth to avoid SAFER+' being used for encryption [6, p. 1677].

We implemented in Python both SAFER+ and SAFER+' including the round computations and the key scheduling algorithm. We tested the two against the specification of Bluetooth (where they are indicated with A_r and A_r' [6, p. 1676]). We also implemented the E and O blocks from Figure 7. The E block is an extension block that transforms the 12 byte COF into a 16 byte sequence using modular arithmetic. The same block is 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. We implement the E and O blocks in Python and we tested them against the specification of Bluetooth. Then, we were able to implement H and to use it to implement and test E_3 and E_1.

We validate the brute forced K'_C by using the necessary parameters from Figure 6 to compute K'_C from K_L. We captured the parameters using the Bluetooth logging capabilities offered by Android. Table 2 shows an example of actual public and private values used during one of our KNOB attacks. We

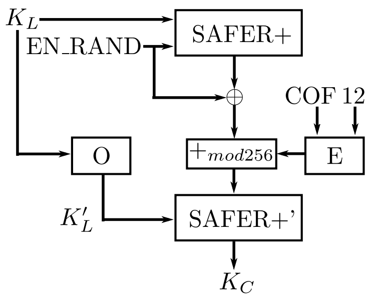

Figure 7: Bluetooth defines H a custom hash function based on SAFER+. H is used to compute K_C from K_L, EN_RAND, and COF (see Equation E_3).

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

plan to release our code implementing E_s, E_1 and E_3 as open-source to help researchers interested in Bluetooth's security, after we complete the responsible disclosure of our findings[1].

4.4 Implementation for Secure Connections

The specification of Bluetooth allows to perform the KNOB attack even when the victims are using Secure Connections. We already implemented the entropy reduction function of the brute force script over AES–CCM. However, at the time of writing, InternalBlue 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.

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

5 Evaluation

Our implementation of the KNOB attack (presented in Section 4) allows to test if any device accepts an encryption key with 1 byte of entropy ($N = L_{min} = 1$). We focus our discussion on the attack best case (1 byte of entropy) while arguably any entropy value lower than 14 bytes could be considered not secure for symmetric encryption [3].

After successfully conducting the KNOB attack on a Nexus 5 and a Motorola G3 we conducted other KNOB attacks on more than 14 unique Bluetooth chips (by attacking 21 different devices). Each attack is easy to reproduce and testing if a device is vulnerable is a matter of seconds.

Based on our experiments, we concluded 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 only 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 tested are not using it.

5.1 Evaluation Setup

To perform our evaluation we collected as many devices as possible containing different Bluetooth chips. At the time of writing, we were able to test chips from Broadcom, Qualcomm, Apple, Intel, and Chicony manufacturers. For each chip we conducted the KNOB attack following the same five steps presented in Section 4.2. As explained earlier, the Nexus 5 is used as a (remote) attacker and a victim. For each test we recorded the manipulated encryption key negotiation protocol over LMP in a pcapng file and we manually verified the protocol's outcome with Wireshark.

Our evaluation setup is not hard to reproduce and easy to extend because it does not require expensive hardware and uses open-source software. We would like to see other researchers evaluating more Bluetooth chips and devices that currently we do not posses, e.g., Apple Watches.

5.2 Evaluation Results

Table 3 shows our evaluation results. Overall, we tested more than 14 Bluetooth chips and 21 devices. The first column contains the Bluetooth chip names. We fill the entries of this

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	✓
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	✓
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 3: List of Bluetooth chips and devices tested against the KNOB attack. ✓indicates that a chip accepts one byte of entropy. * indicates that a chip accepts at least seven bytes of entropy. We note that, all chips and devices implementing any specification of Bluetooth are expected to be vulnerable to the KNOB attack because the entropy reduction feature is standard-compliant.

column with Unknown when we are not able to find information about the chip manufacturer and/or model number. The second column lists the devices that we tested 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. The table's rows are grouped by Bluetooth version in four blocks: version 5.0, version 4.2, version 4,1 and version lower or equal than 4.0.

From the third column of Table 3 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. Seven bytes of entropy are better than one, but not enough to prevent brute force attacks. For example, the Data Encryption Standard (DES) uses the same amount of entropy and DES keys were brute forced multiple times with increasing efficacy [19].

Table 3 also demonstrates that the vulnerability spans across different Bluetooth versions including the latest ones such as 5.0 and 4.2. This fact confirms that the KNOB attack is a significant threat for all Bluetooth users and we believe that the specification of Bluetooth has to be fixed as soon as possible.

6 Discussion

6.1 Attacking Other Bluetooth Profiles

Cable replacement wireless technologies such as Bluetooth are widely used for all sorts of applications including desktop, mobile, IoT, industrial and medical devices. Bluetooth defines its set of application layer services as profiles. In Section 4 we describe an attack on the OBject EXchange (OBEX) Bluetooth profile, where the attacker breaks Bluetooth security by decrypting the content of an encrypted file without having access to any (pre-shared) secret. Here we describe three KNOB attacks targeting other popular Bluetooth profiles. As in the OBEX case, the attacks have serious implications in terms of security and privacy of the victims. To the best of our knowledge, all the profiles that we discuss in this section rely only on the link-layer for their security guarantees and they are widely used across different vendors. Our list of attacks is not exhaustive and an attacker might exploit the vulnerable encryption key negotiation protocol of Bluetooth in other creative ways.

HID profile The attacker could perform a remote keylogging attack on any device that uses the Human Interface Device (HID) profile. This profile is used by input-output devices such as keyboards, mice and joysticks. As a result, the attacker can sniff sensitive information including passwords, credit card numbers, and emails regardless if these information are then encrypted on the (wired or wireless) Ethernet link.

Bluetooth tethering The attacker could mount a remote man-in-the-middle attack when the victim uses Bluetooth for tethering. Tethering is used by a device, acting as an hotspot, to share Internet connectivity with other devices in range. Bluetooth transports Ethernet over the Bluetooth Network Encapsulation Protocol (BNEP) [5]. This protocol encapsulates Ethernet frames and transports them over (link-layer encrypted) L2CAP. As a result, the attacker can sniff all Internet traffic of the victims using a Bluetooth hotspot.

A2DP profile The attacker could record and inject audio signals when the victim uses the Advanced Audio Distribution Profile (A2DP) profile. As a result, the attacker is able to record phone and Voice over IP (VoIP) calls even if the call is encrypted (e.g., 4G and Skype). The attacker can also tamper with voice commands sent to a personal assistant, e.g., Siri and Google Assistant. Recent mobile devices, such as smartphone and tablets, are particularly vulnerable to this threat because Bluetooth is a convenient solution to the lack of an analog audio connector (audio jack).

6.2 Attacking Multiple Nodes and Piconets

In our paper we describe the implementation of KNOB attacks targeting two victims. If a Bluetooth piconet contains more than two devices, then (in the worst case for the attacker) each master-slave pair uses a dedicated set of keys. In this scenario the KNOB attack still works because it can be parallelized with minimal effort. For example, the attacker may run the same attack script on different computing units, such as processes or machines, and let each computing unit target a master-slave pair. Each parallel instance of the attack negotiates an encryption key with one byte of entropy, captures the exchanged ciphertext, and brute forces the encryption key. For example, an attacker is able to decrypt all the traffic from a victim using multiple Bluetooth I/O devices to interact with his device e.g., a laptop connected with a keyboard, a mouse and an headset.

The KNOB attack is effective even if the attacker wants to target multiple piconets (Bluetooth networks) at the same time. In this case the attacker has to follow and use a different Bluetooth clock (CLK) value for each piconet to compute the correct encryption key. This is not a problem because the attacker can use parallel KNOB attack instances, where each instance follows a pair of devices in a target piconet.

6.3 Practical Implementation Issues

We spent considerable time to fine tune our brute force script. One main reason is that Ubertooth One, used to sniff Bluetooth BR/EDR packets over the air, does not reliably capture all packets and clock values (CLK). This is true even if we limit our capture to a specific piconet by setting the UAP and LAP parameters. As a result, we had to include extra logic in our brute force script to iterate over different CLK values and E_0 keystream offsets. Our basic brute force logic only iterates over the encryption key space (256 iterations). The extra logic can be removed if we get access to a commercial-grade Bluetooth protocol analyzer such as Ellisys [10] or similar. Unfortunately, these devices are very expensive.

We implemented our attack by simulating a remote attacker using InternalBlue. Alternatively, we could have conducted the attacks over the air using signal manipulation [26] and (reactive) jamming [31]. However, the InternalBlue setup is simpler, more reliable, cheaper, and easier to reproduce than the over-the-air setup and it affects the victims in the same way as a remote attacker.

6.4 Countermeasures

In this section we propose several countermeasures to the KNOB attack. We divide them into two classes: legacy compliant and non legacy compliant. The former type of countermeasure does not require a change to the specification of Bluetooth while the latter does. We already proposed these countermeasures to the Bluetooth SIG and CERT during our responsible disclosure.

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), not as part of the entropy negotiation protocol. A third solution is to distrust the link layer and provide the security guarantees at the application layer. Some vendors have done so by adding a custom application layer security mechanism on top of Bluetooth (which, in case of Google Nearby Connections, was also found to be vulnerable [1]).

Non legacy compliant. A non legacy compliant countermeasure is to modify the encryption key negotiation protocol by securing it using the link key. The link key is a shared (and possibly authenticated) secret that should be always available before starting the entropy negotiation protocol. The new protocol should provide message integrity and might also provide confidentiality. 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 solutions only requires the modification of the Bluetooth controller (firmware).

7 Related Work

The security and privacy guarantees of Bluetooth were studied since Bluetooth v1.0 [15, 32]. Particular attention was given to Secure Simple Pairing (SSP), a mechanisms that Bluetooth uses to generate and share a long term secret (defined as the link key). Several attacks on the SSP protocol were proposed [28, 13, 4]. The Key Negotiation Of Bluetooth (KNOB) attack works regardless of security guarantees provided by SSP (such as mutual user authentication).

The most up to date survey about Bluetooth security was provided by NIST in 2017 [25]. This survey recommends to use 128 bit keys (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 [9] or superficially discuss it [29].

The various implementation of Bluetooth were also analyzed and several attacks were presented on Android, iOS, Windows and Linux implementations [2]. Our 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 has been extensively discussed by cryptographers. The SAFER+ cipher used by Bluetooth for authentication purposes was analyzed [17]. The E_0 cipher used by Bluetooth for encryption was also analyzed [11]. Our attack works even with perfectly secure ciphers. For our implementation of the custom Bluetooth security procedures (presented in Section 4) we used as main references the specification of Bluetooth [6] and third-party hardware [18] and software [20] implementations.

Third-party manipulations of key negotiation protocols were also discussed in the context of WiFi, for example key reuse in [30]. Compared to those attacks, our attack exploits not only implementation issues, but a standard-compliant vulnerability of the specification of Bluetooth.

Protocol downgrade attacks were discussed in the context of TLS[23], where the two parties are negotiating the cipher suite to use. We note that in contrast to our scenario, for TLS the application developers have commonly direct control over the cipher suites that will be offered by their applications. Therefore, avoiding a fallback to legacy encryption standards can be prevented by the developers. To the best of our knowledge, this is not the case for Bluetooth, as the protocols does not enforce any mandatory checks on the encryption key's entropy.

8 Conclusion

In this paper we present the Key Negotiation Of Bluetooth (KNOB) attack. Our attack is capable of reducing the entropy of the encryption key of any Bluetooth BR/EDR connection to 1 byte (8 bits). The attack is standard-compliant because the specification of Bluetooth includes an insecure encryption key negotiation protocol that supports entropy values between 1 and 16 bytes. As a main consequence, an attacker can easily negotiate an encryption key with low entropy and then brute force it. The attacker is effectively breaking the security guarantees of Bluetooth without having to posses any (pre-shared) secret material. The attack is stealthy because the vulnerable entropy negotiation protocol is run by the victims' Bluetooth controller and this protocol is transparent to the Bluetooth host (OS) and the Bluetooth application used by the victims. We expect that the attack could be run in parallel to target multiple devices and piconets at the same time.

We demonstrate that the KNOB attack can be performed in practice by implementing it to attack a Nexus 5 and a Motorola G3. In our attack we decrypt a file transmitted over an authenticated and link-layer encrypted Bluetooth connection. Brute-forcing a key with 1 byte of entropy introduces a negligible overhead enabling an attacker to decrypt all the ciphertext and to introduce valid ciphertext even in real-time.

We evaluate the KNOB attack on more than 14 Bluetooth chips from different vendors such as Broadcom, Qualcomm and Intel. All the chips accept 1 byte of entropy except the Apple W1 chip that accepts (at least) 7 bytes of entropy. Frankly, we were expecting to find more non standard-compliant chips like the Apple W1. Before submitting the paper, we reported our findings to the Computer Emergency Response Team (CERT) and the Bluetooth Special Interest Group (SIG). Both organizations acknowledged the problem and we are collaborating with them to solve it. After our responsible disclosure, we plan to release the tools that we developed to implement the attacks as open-source.

The KNOB attack is a serious threat to the security and privacy of all Bluetooth users. We were surprised to discover such fundamental issues in a widely used and 20 years old standard. We attribute the identified issues in part to ambiguous phrasing in the standard, as it is not clear who is responsible for enforcing the entropy of the encryption keys, and as a result no-one seems to be responsible in practice. We urge the Bluetooth SIG to update the specification of Bluetooth according to our findings. Until the specification is not fixed, we do not recommend to trust any link-layer encrypted Bluetooth BR/EDR link. In Section 6.4 we propose legacy and non legacy compliant countermeasures that would make the KNOB attack impractical. We also recommend the Bluetooth SIG to create a dedicated procedure enabling researchers to securely submit new potential vulnerabilities, similarly to what other companies, such as Google, Microsoft and Facebook, are offering.

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

[3] Elaine Barker, William Barker, William Burr, William Polk, and Miles Smid. Recommendation for key man-

agement part 1: General (revision 3). *NIST special publication*, 800(57):1–147, 2012.

[4] 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.

[5] Bluetooth SIG. Bluetooth Network Encapsulation Protocol. `http://grouper.ieee.org/groups/802/15/Bluetooth/BNEP.pdf`, Accessed: 2018-10-28, 2001.

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

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

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

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

[10] Ellisys. Ellisys protocol test solutions. `https://www.ellisys.com/`, Accessed: 2018-10-28.

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

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

[13] 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.

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

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

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

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

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

[19] 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.

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

[21] 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.

[22] 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.

[23] 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.

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

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

[26] 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.

[27] 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.

[28] 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.

[29] 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.

[30] 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.

[31] 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.

[32] 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.

A Appendix

The Key Negotiation Of Bluetooth (KNOB) attack reduces the entropy of the encryption key (K'_C) to 1 byte (key space has 256 elements). Table 4 shows twenty encryption keys with one byte of entropy both for E_0 and AES-CCM.

E_0 K'_C in hex, MSB on the left	AES-CCM K'_C in hex, MSB on the left
`0x00000000000000000000000000000000`	`0x00000000000000000000000000000000`
`0x00e275a0abd218d4cf928b9bbf6cb08f`	`0x01000000000000000000000000000000`
`0x01c4eb4157a431a99f2517377ed9611e`	`0x02000000000000000000000000000000`
`0x01269ee1fc76297d50b79cacc1b5d191`	`0x03000000000000000000000000000000`
`0x0389d682af4863533e4a2e6efdb2c23c`	`0x04000000000000000000000000000000`
`0x036ba322049a7b87f1d8a5f542de72b3`	`0x05000000000000000000000000000000`
`0x024d3dc3f8ec52faa16f3959836ba322`	`0x06000000000000000000000000000000`
`0x02af4863533e4a2e6efdb2c23c0713ad`	`0x07000000000000000000000000000000`
`0x0713ad055e90c6a67c945cddfb658478`	`0x08000000000000000000000000000000`
`0x07f1d8a5f542de72b306d746440934f7`	`0x09000000000000000000000000000000`
`0x06d746440934f70fe3b14bea85bce566`	`0x0a000000000000000000000000000000`
`0x063533e4a2e6efdb2c23c0713ad055e9`	`0x0b000000000000000000000000000000`
`0x049a7b87f1d8a5f542de72b306d74644`	`0x0c000000000000000000000000000000`
`0x04780e275a0abd218d4cf928b9bbf6cb`	`0x0d000000000000000000000000000000`
`0x055e90c6a67c945cddfb6584780e275a`	`0x0e000000000000000000000000000000`
`0x05bce5660dae8c881269ee1fc76297d5`	`0x0f000000000000000000000000000000`
`0x0e275a0abd218d4cf928b9bbf6cb08f0`	`0x10000000000000000000000000000000`
`0x0ec52faa16f3959836ba322049a7b87f`	`0x11000000000000000000000000000000`
`0x0fe3b14bea85bce5660dae8c881269ee`	`0x12000000000000000000000000000000`
`0x0f01c4eb4157a431a99f2517377ed961`	`0x13000000000000000000000000000000`

Table 4: List of twenty K'_C used by E_0 (left column) and AES-CCM (right column) when $N = 1$ (key space is 256).