Bypassing Passkey Authentication in Bluetooth Low Energy

(extended abstract)

Tomáš Rosa

tomas.rosa@rb.cz

Raiffeisenbank, a.s.
Hvězdova 1716/2b, 140 78 Praha 4

Keywords: Bluetooth Low Energy (Smart), Security Manager, Authentication, Cryptanalysis

This memo describes certain new cryptographic weakness of the passkey-based pairing of Bluetooth LE (BLE or BTLE, also known as *Bluetooth Smart*; as one prefers). The vulnerability discussed here extends the set of possible attacking scenarios that were already elaborated before by Mike Ryan in [4].

Instead of the passive sniffing attack on pairing secrets, we show how a fraudulent *Responder* [1] can gracefully bypass the passkey authentication, despite it being possibly based on even one-time generated PIN.

Such an active attack may become handy in situation where passive sniffing of correct pairing cannot be employed – for instance, because of the original *Responder* device being out of reach or otherwise unwilling to pair again. Or, we may already want to actively impersonate the peripheral device to inject some data into e.g. iPhone Apps from attacker's keyboard, perform MITM, etc.

Since the attack runs on the *Security Manager* layer [1], it can reuse a lot of the existing network stack that is already in place for this approach. This namely concerns everything bellow *Host Controller Interface* (HCI) [1]. Actually, the whole procedure starting with the authentication bypass and continuing to data injection (which would be a regular communication anyway) can be done using a general *Bluetooth 4.0 Smart Ready* USB dongle via HCI commands.

Furthermore, we shall perhaps emphasize the attack we present here would be possible even if there already was the yet-awaited ephemeral Diffie-Hellman key agreement employed in BLE as, for instance, in Bluetooth BR/EDR *Secure Simple Pairing* [1]. In other words, introducing D-H would not prevent this attack as long as the function c_1 (cf. bellow) would not be redesigned as well.

Flawed Bit Commitment

The flaw was discovered during an investigation of cryptographic properties of the *Bit Commitment* protocols in the way they are employed in the authentication schemes of Wi-Fi *Protected Setup*, BT *Secure Simple Pairing*, and BLE *Security Manager*. Despite targeting different radio networks, they share a lot of common ideas, namely the mutual authentication based on *Bit Commitment* variants [2].

The notation bellow follows the one used in Bluetooth Core Spec. v 4.0 [1].

In particular, we address the "*Confirm value generation function*" denoted c_1 in [1], Vol. 3, part H-2.2.3. This function actually computes a commitment of the respective party (*Initiator* or *Responder*) to a secret passkey together with labels p_1, p_2 related to the actual public pairing parameters. In particular, for the passkey authentication commitment value C, we have:

$$C = c_1(TK, rand, p_1, p_2) = \mathrm{AES}_{TK}[\mathrm{AES}_{TK}(rand \oplus p_1) \oplus p_2], \quad \text{(Eq. 1)}$$

where:

- *TK* (*Temporary Key*) is the passkey derived directly from the common 6-digit PIN (the six-digit length is constant in BLE),
- *rand* is 128 bits long secret value [1].

Actually, the function c_1 servers the role of the *Bit Commitment* primitive called "*Commit*" with (TK, p_1, p_2) being the committed message and *rand* being the opener [5].

We show the function c_1 lacks so-called binding property, so the originator can freely change the committed message even after having already announced C, provided they did not reveal the particular *rand*, yet. We will further show this notable weakness allows a dishonest *Responder* to bypass the authentication procedure even for a one-time PIN.

We start by showing how to change the "committed" message. Let us be given any value of commitment C and let (TK, p_1, p_2) be arbitrarily chosen after C has been already announced. Then we can trivially find a new valid *rand* (the opener) as:

$$rand = \mathrm{AES}^{-1}_{TK}[\mathrm{AES}^{-1}_{TK}(C) \oplus p_2] \oplus p_1. \qquad \text{(Eq. 2)}$$

Proof. Substituting this value of *rand* into the original Eq. 1, we can verify that indeed

$$\begin{aligned} C &= c_1(TK, rand, p_1, p_2) \\ &= \mathrm{AES}_{TK}[\mathrm{AES}_{TK}(rand \oplus p_1) \oplus p_2] \\ &= \mathrm{AES}_{TK}[\mathrm{AES}_{TK}(\mathrm{AES}^{-1}_{TK}[\mathrm{AES}^{-1}_{TK}(C) \oplus p_2] \oplus p_1 \oplus p_1) \oplus p_2] \\ &= \mathrm{AES}_{TK}[\mathrm{AES}_{TK}(\mathrm{AES}^{-1}_{TK}[\mathrm{AES}^{-1}_{TK}(C) \oplus p_2]) \oplus p_2] \\ &= \mathrm{AES}_{TK}[\mathrm{AES}^{-1}_{TK}(C) \oplus p_2 \oplus p_2] \\ &= \mathrm{AES}_{TK}[\mathrm{AES}^{-1}_{TK}(C)] \\ &= C \end{aligned}$$

□

Subverting the Authentication Protocol

To see a practical application of the c_1 weakness discussed above, let us consider a *Responder* (in the context of BLE *Security Manager* [1]), who has already sent their commitment *Sconfirm*,

$$Sconfirm =_{presumably} c_1(TK, Srand, p_1, p_2) = \mathrm{AES}_{TK}[\mathrm{AES}_{TK}(Srand \oplus p_1) \oplus p_2],$$

but who has not revealed their *Srand*, yet.

Due to the lack of binding in c_1, such a *Responder* can still arbitrarily change their "committed" passkey TK and labels p_1, p_2, since – as we have seen above – the correct *Srand* for any new value of (TK, p_1, p_2) can be trivially found by Eq. 2 while keeping the former *Sconfirm* still the same.

This implies that even the one-time passkey – i.e. a fresh value of PIN generated for each and every single pairing protocol run – can be easily broken by a fraudulent *Responder*.

The illustrative attack procedure follows (cf. [1], Vol. 3, part H-2.3.5.5 for the context):

i) *Initiator* sends *Mconfirm* to the *Responder* (i.e. to the attacker)

ii) the attacker (as *Responder*) responds with a purely random value of *Sconfirm*

iii) *Initiator* sends *Mrand*, such that $Mconfirm = c_1(TK, Mrand, p_1, p_2) = \mathrm{AES}_{TK}[\mathrm{AES}_{TK}(Mrand \oplus p_1) \oplus p_2]$

iv) using a brute-force, the attacker finds the correct passkey TK (based on a 6-digit PIN) from *Mconfirm*, *Mrand*, and known labels p_1, p_2 (already noted in a different attack in [4], [1])

v) having gained the correct passkey TK, the attacker uses Eq. 2 to compute its corresponding correct value of *Srand* and sends it to *Initiator*

vi) *Initiator* receives the (just corrected) *Srand* and using the original Eq. 1 verifies that it indeed corresponds with *Sconfirm* received in step (ii) before, so the *Initiator* now believes the right passkey was already known to the *Responder* before step (iii)!

vii) *Initiator* concludes the passkey was verified successfully and continues with *STK* derivation and so on [1]

Now, the attacker knows everything needed to derive the correct *Short Term Key* (*STK*) [1] and is fully in the position to follow the pairing procedure to its "happy end". Actually, the whole pairing procedure is going right according to the standard [1] with just one imperfection – the *Responder*'s part of the authentication has been bypassed.

BLE MITM Security Assurance Is False

We have just seen the *Passkey Entry* pairing method of the *Bluetooth Low Energy* communication standard fails to provide authentication of the *Responder* to the *Initiator* even with a one-time generated PIN. There is at most a one-way authentication of the *Initiator* to the *Responder* achieved, provided the attacker cannot mount the MITM noted bellow for some reason (for instance, because there is no original Initiator available).

The elaboration given above shows the conjecture noted in [1], Vol.3, part H-2.3.5.3, saying: "*...The passkey Entry method provides protection against active "man-in-the-middle" (MITM) attacks as an active man-in-the-middle will succeed with a probability of 0.000001 on each invocation of the method...*", is false.

From a cryptography viewpoint, we have just seen an active MITM (attacker in between the honest *Initiator* and *Responder*) succeeds with probability 1. The active attacker would first use the aforementioned procedure to authenticate with the honest *Initiator*. After having established this link together with having learned the correct passkey *TK*, the attacker starts its own pairing procedure with the honest *Responder*. Note that, since this is the very first interaction with the honest *Responder* (the previous interaction in between the *Initiator* and the *di*shonest *Responder* did not concern the honest one), it will still regard this value of *TK* as a valid one even under the possible one-time PIN policy.

Simple Python code [3] was written to verify the idea is mathematically correct in that sense it provides us with the correct *Srand* as needed. It would be interesting to see its practical applications and further extensions.

Countermeasures
Interestingly, under a reasonable assumption that *Srand* is the only commitment-related value the attacker can change after having sent *Sconfirm*, there are several trivial hot fixes possible. Basically, all we need is to perform just one more eXclusive OR (xor) operation.

The main idea of the countermeasure is to disrupt the unwanted reversibility of c_1 towards *Srand* under fixed (p_1, p_2). This can be practically achieved by any one of the following constructions of "c_1-*fixed*":

a) $c_1\text{-}fixed(TK, rand, p_1, p_2) = \mathrm{AES}_{TK \oplus \mathbf{rand}}[\mathrm{AES}_{TK \oplus \mathbf{rand}}(rand \oplus p_1) \oplus p_2]$

b) $c_1\text{-}fixed(TK, rand, p_1, p_2) = \mathrm{AES}_{TK}[\mathrm{AES}_{TK}(rand \oplus p_1) \oplus \mathbf{rand} \oplus p_2]$

c) $c_1\text{-}fixed(TK, rand, p_1, p_2) = \mathrm{AES}_{TK}[\mathrm{AES}_{TK}(rand \oplus p_1) \oplus p_2] \oplus \mathbf{rand}$

Recall this hot-fix does rely on (p_1, p_2) being invariable after the commitment has been made – i.e. after the *Responder* has sent its *Sconfirm* to the *Initiator*. Fortunately, this is true for the *Passkey Entry* pairing protocol in BLE, so the patch can be applied. Furthermore, it can be seen as being aligned with the whole BLE strategy – to do exactly what is necessary, no less no more.

Since the countermeasure is to be done at *Security Manager* layer, it does not affect the Bluetooth controller firmware bellow HCI.

References

[1] *Bluetooth Core Specification*, ver. 4.0, Bluetooth SIG, June 2010

[2] Rosa, T.: *Wi-Fi Protected Setup - Friend or Foe*, Smart Cards & Devices Forum, Prague, May 23rd, 2013, http://crypto.hyperlink.cz/files/rosa_scadforum13.pdf

[3] http://crypto.hyperlink.cz/files/blecommit.py

[4] Ryan, M.: *How Smart is Bluetooth Smart?*, Shmoocon 2013, Feb 16th, http://lacklustre.net/bluetooth/ [*retrieved May-16-2013*]

[5] http://en.wikipedia.org/wiki/Bit_commitment [*retrieved Nov-12-2013*]