Bypassing Passkey Authentication in Bluetooth Low Energy

(extended abstract, May 16th 2013)

Tomáš Rosa

http://crypto.hyperlink.cz

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

Accompanying presentation: http://crypto.hyperlink.cz/files/rosa_scadforum13.pdf [2]

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 can gracefully bypass 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 the original Responder device is 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, perform MITM, etc.

Since the attack runs on the Security Manager layer, it can reuse a lot of the existing network stack that is already in place for this approach. This namely concerns everything bellow 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 if the function c_1 (cf. bellow) would not be redesigned as well.

Now back to the main crypto part. This flaw was discovered during the investigation of cryptographic properties of the Bit Commitment protocols 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. Here, it 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 the passkey TK is directly derived from the 6-digit PIN (constant length of 6 digits) and $rand$ is 128 bits long yet-secret value [1]. Basically, c_1 resembles a Bit Commitment primitive here where (TK, p_1, p_2) is the committed message and $rand$ is the opener [2].

It can be shown (cf. bellow and [2], slides 35 to 39) that the function c_1 lacks so-called binding property (cf. [2], slide 18), which is a notable weakness here. 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. We can trivially find a new valid $rand$ (opener) as:

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

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

$C = c_1(TK, rand, p_1, p_2) =$

$= \mathrm{AES}_{TK}[\mathrm{AES}_{TK}(rand \oplus p_1) \oplus p_2] =$

$= \mathrm{AES}_{TK}[\mathrm{AES}_{TK}(\mathrm{AES}^{-1}{}_{TK}[\mathrm{AES}^{-1}{}_{TK}(C) \oplus p_2] \oplus p_1 \oplus p_1) \oplus p_2] =$

$= \mathrm{AES}_{TK}[\mathrm{AES}_{TK}(\mathrm{AES}^{-1}{}_{TK}[\mathrm{AES}^{-1}{}_{TK}(C) \oplus p_2]) \oplus p_2] =$

$= \mathrm{AES}_{TK}[\mathrm{AES}^{-1}{}_{TK}(C) \oplus p_2 \oplus p_2] =$

$= \mathrm{AES}_{TK}[\mathrm{AES}^{-1}{}_{TK}(C)] =$

$= C$

□

To see a practical application of this c_1 weakness, 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 attack procedure is this (cf. [1], Vol. 3, part H-2.3.5.5 for the context):

i) Initiator sends *Mconfirm* to the Responder (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 [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 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 correct *STK* 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 authentication has been bypassed.

In other words, the Passkey Entry pairing method of Bluetooth Low Energy 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 here 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.

We have just seen an active MITM (attacker in between the honest Initiator and Responder) succeeds with probability 1 (from the cryptography viewpoint). The active attacker would first use the aforementioned procedure to authenticate with the honest Initiator. After having learned the correct passkey *TK*, the attacker starts its own pairing procedure with the honest Responder.

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.

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 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 achieved by any one of the following modifications:

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

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

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

Recall this fix does rely on (p_1, p_2) being fixed after the commitment has been made (i.e. after Responder has sent *Sconfirm* to Initiator). On the other hand, this is true for Passkey Entry pairing protocol in BLE, so the fix can be seen as being aligned with the whole BLE strategy – to do exactly what is necessary, no less no more. Since the fix 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*]