Analysis of eQ-3 BLUETOOTH® Smart Lock

Introduction

In this project we reverse engineered the Android application for the eQ-3 Bluetooth Smart Lock in order to understand the protocol, check if we can find any security flaws and develop a compatible open source program for controlling the lock from a Linux computer.
The app is available from the Google Play Store and was extracted from a smartphone after download. The Java classes are partially obfuscated.

Hardware

The device has two processors: A Broadcom BCM20736S bluetooth SOC containing a Cortex-M3 ARM core, and a STMicroelectronics STM8L052 microcontroller which is based on the proprietary STM8 architecture. The whole application logic is implemented in the STM8 controller, whereas the Broadcom SOC handles the Bluetooth connection.

Illustration 1: Photo of the microprocessors

Inter MCU Communication

The Broadcom controller handles the Bluetooth communication with the smart phone, including the fragmentation and defragmentation of data packets in 15-byte fragments for transmission via the Bluetooth GATT protocol. Between the controllers, a UART connection with 117000 baud is used to exchange unfragmented data packets received from the phone, and to be sent back to the phone. A custom protocol is used for framing and error control. The packet format is described in pseudo-C[1] as follows:

```
struct uart_packet {
        char preamble[3]; // fixed { 0xf0, 0xf0, 0xfd }
        int16_t length;
        char data[length – 6];  //contains full packets (command type + payload)
        char bluetooth_mac_address[6];
        char checksum[2];
}
```

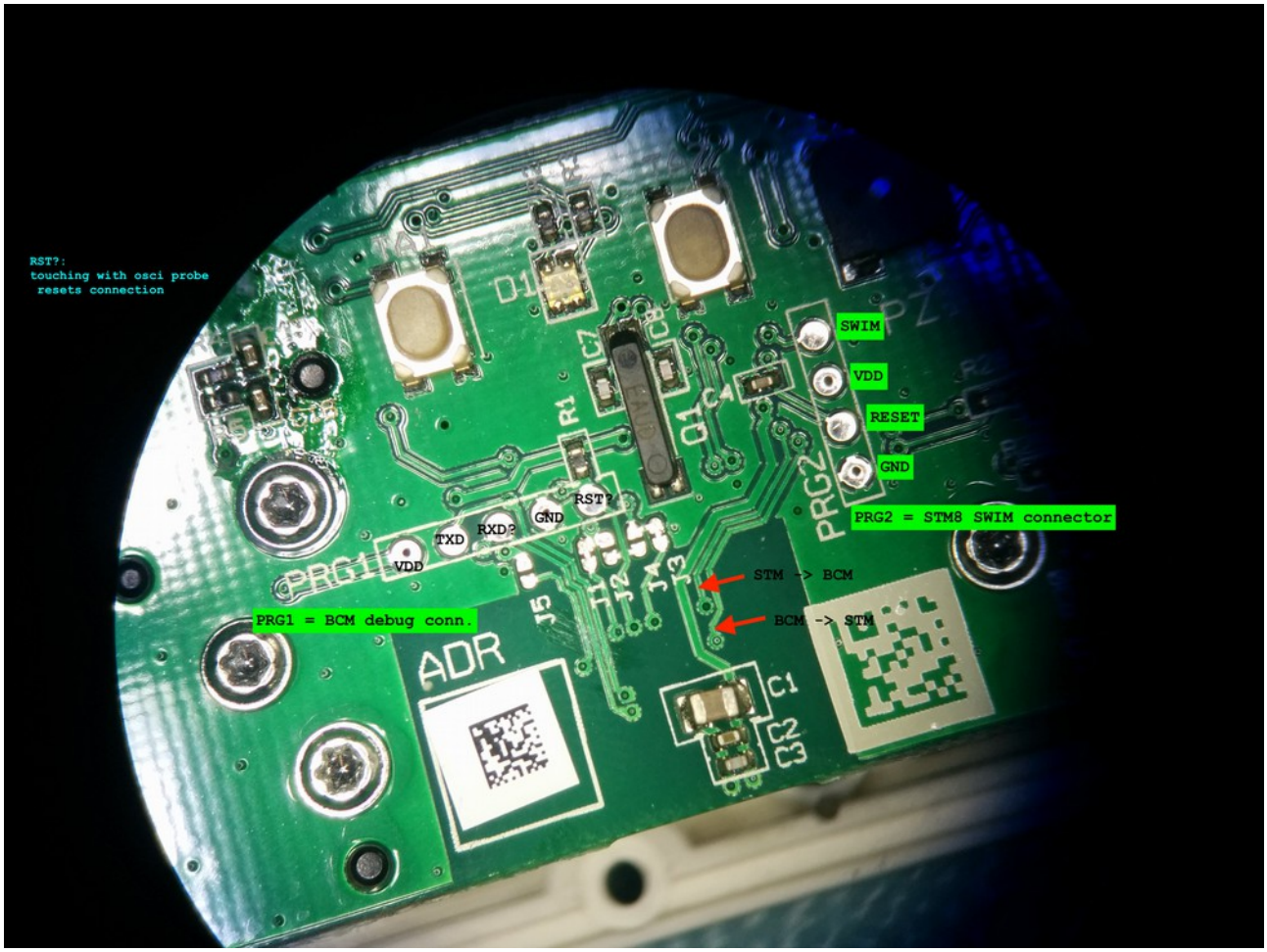

Illustration 2: Location of debug pins on the PCB

1 All pseudo-C struct in this document should be interpreted as „packed“, meaning no word alignment takes place. Also, the signedness of fields was not checked in most cases.

Firmware

We were unable to extract the firmware of the STM8 microcontroller as the readout protection was enabled and the integrated ROM bootloader was disabled. If the bootloader had been enabled, a readout would likely have been possible due to a publicly known bug in the bootloader where the readout protection bits are ignored.

STM8 group	Bootloader version	Device revision	Limitations, improvements, and added features
	1.0	Rev A	Initial version Limitations: – Readout protection option bit (ROP) not checked – CPU clock is set to HSI/1 (16 MHz) when bootloader resumes – Timeout for SYNCH byte receiving after reset is 500 ms instead of 1 second.

Illustration 3: Excerpt from ST UM0560, Limitations and improvements versus bootloader versions

A binary firmware update blob for the STM8 is distributed in the Android application (res/raw/update.eq3). When an update is sent to the device, it is prepared by the app as follows: Read the whole file as a string, decode it as a hex values into a byte array. Parse the byte array into chunks of the following format:

```
struct update_eq3_chunk {
        int16_t chunk_length;
        char data[chunk_length];
        int16_t checksum;
}
```

A list of byte arrays containing only the data portions of each chunk is generated. The checksums seem to be ignored. Each chunk is sent as a packet with command type BOOTLOADER_DATA.

We parsed the file and stored the data portions concatenated. However, the entropy of the result as measured by binwalk is >99%, and disassembling it with the STM8 disassembler yields many garbage instructions. No strings are found in it. Therefore we assume it to be compressed and/or encrypted in an unknown manner. We didn‘t pursue this any further and concentrated on the protocol as implemented in the Java classes.

There is another firmware file distributed (res/raw/update_ota.bin), which, going by the included strings, seems to be for updating the Broadcom controller. However, we didn‘t find blocks of valid ARM 32-bit instructions in the non-string parts. It might however be code in the ARM Thumb instruction set. As this controller doesn‘t handle any application specific behavior, but only standard bluetooth connection, we also didn‘t investigate the file further.

Deobfuscating the app

The Android app is only lightly obfuscated. Most UI parts and libraries are not obfuscated at all. Only the packages implementing the protocol where obfuscated in a way where all package, class

and member names where replaced by letters a, b, c and so on. Because method overloading was used such that many methods where named a in a class, differing only in parameter count and type, simple search and replace was unusable for deobfuscation, also the refactoring tool of Eclipse failed in many cases. The binary-refactor[2] software was very useful, because it works directly on the JAR file, using internal references for renaming classes, methods and fields, and can‘t be tricked by methods with identical names. Also some helpful strings and an unobfuscated enumeration of command types where still present in the app.

Bluetooth standard protocols

For communication between the app and the bluetooth chip, standard Bluetooth Low Energy is used, with the Bluetooth Attribute Protocol on top. The service and characteristics UUIDs over which the application protocol is sent, are listed below (described from the view of the Android app):

Service: 58e06900-15d8-11e6-b737-0002a5d5c51b
Write Characteristic (client to lock): 3141dd40-15db-11e6-a24b-0002a5d5c51b
Read Characteristic (lock to client): 359d4820-15db-11e6-82bd-0002a5d5c51b

For sending commands to the lock, a Write Request is sent on the Write Characteristic. Replies and notifications by the lock are received as Handle Value Notifications on the Read Characteristic.

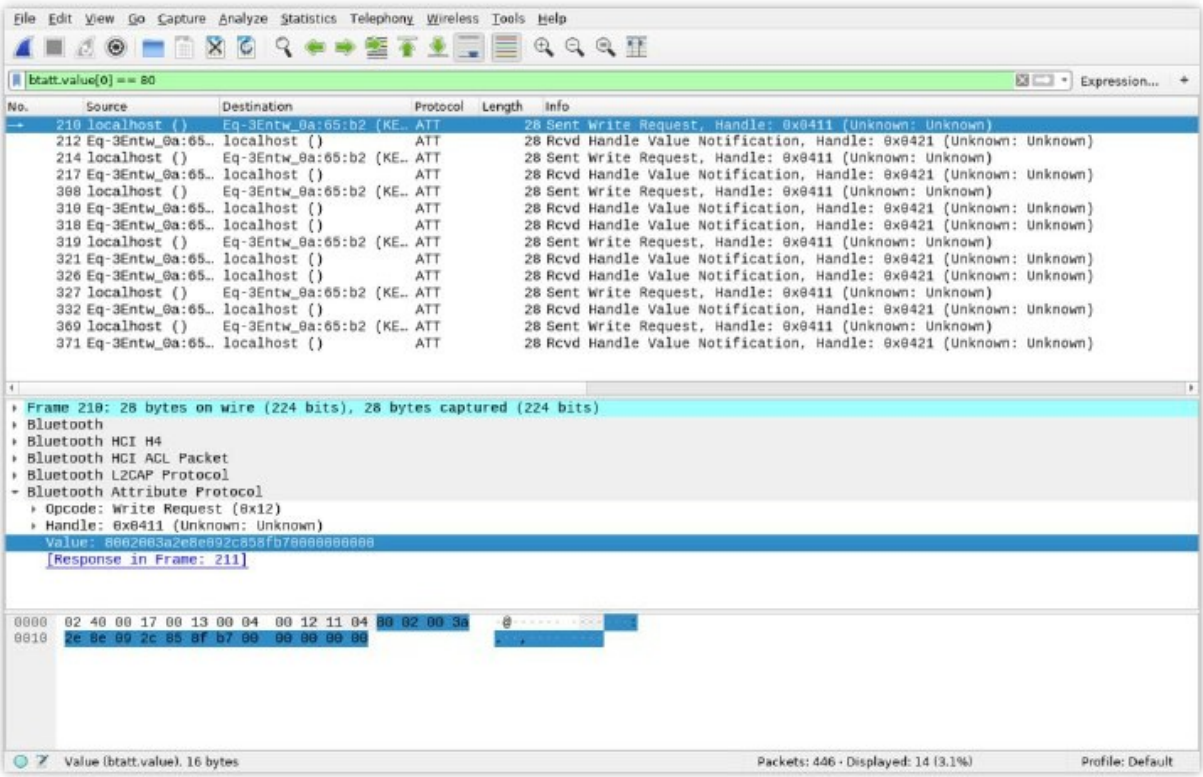

Illustration 4: Packet capture of bluetooth connection, the highlighted packet is a CONNECTION_REQUEST

2 https://github.com/argan/binary-refactor

Transport protocol

A custom transport layer protocol is used for fragmenting data, as Bluetooth Attribute Protocol frames have a limited length. The transport protocol produces fragments of at most 16 byte length, consisting of a one byte header and an up to 15 byte payload. The most significant bit of the header byte is 1 if it is the first fragment of a packet, 0 otherwise. The 7 least significant bits give the number of fragments to come after this one (remaining_frag). For example,

- a packet of under 15 bytes length is just prefixed by 0x80 (= first fragment, 0 fragments to come)
- two fragments are represented like this: 0x81 [15 data bytes]; 0x00 [<= 15 data bytes]
- three fragments: 0x82 [15 data bytes]; 0x01 [15 data bytes]; 0x00 [<= 15 data bytes]

After sending a fragment which is not the last in a packet, the sender waits for a FRAGMENT_ACK packet in response, before transmitting the next fragment. It has one byte payload, containing the header byte of the fragment just received. If a fragment is the last one in the packet, it is not acknowledged.

Application protocol

Command types

Each packet has a command type, which is transferred as the first byte of the packet. If the two uppermost bits of the command type are 0b10 (meaning the command type is between 128 and 191), the command is transmitted „secure“ (authenticated and encrypted), otherwise it is transferred in plain text. For plain text packets, the command type byte is directly followed by payload bytes whose number and meaning is command-specific. The format of secure packets is described in a following section.

Below is a list of all known command types, as extracted from an enumeration in the app.

Unsecured

- FRAGMENT_ACK(0),
- ANSWER_WITHOUT_SECURITY(1),
- CONNECTION_REQUEST(2),

```
struct CONNECTION_REQUEST_COMMAND {
      uint8_t command_type; //0x02
      char userNumber;
      char applicationNonce[8];
}
```

- CONNECTION_INFO(3),

```
struct CONNECTION_INFO_COMMAND {
      uint8_t command_type; //0x03
      char pairingCounter;
      char deviceNonce[8];
      char unknown;
      char bootloaderVersion;
      char applicationVersion;
}
```

- PAIRING_REQUEST(4),
- STATUS_CHANGED_NOTIFICATION(5),
- CLOSE_CONNECTION(6),
- BOOTLOADER_START_APP(16),
- BOOTLOADER_DATA(17),
- BOOTLOADER_STATUS(18),

Secured (128-191)

- ANSWER_WITH_SECURITY(129),
- STATUS_REQUEST(130),

```
struct STATUS_REQUEST_COMMAND {
      uint8_t command_type; //0x82
      char year; //year minus 2000
      char month;
      char day;
      char hour;
      char minute;
      char second;
}
```

- STATUS_INFO(131),

```
struct STATUS_INFO_COMMAND {
      uint8_t command_type; // 0x83
      char controlFlags;    // see pysmartlock implementation
      char statusFlags;     // for flag definitions
      char lockFlags;
      char unknown;
      char bootloaderVersion;
      char applicationVersion;
}
```

- MOUNT_OPTIONS_REQUEST(132),
- MOUNT_OPTIONS_INFO(133),
- MOUNT_OPTIONS_SET(134),
- COMMAND(135),
- AUTO_RELOCK_SET(136),
- PAIRING_SET(138),
- USER_LIST_REQUEST(139),
- USER_LIST_INFO(140),
- USER_REMOVE(141),
- USER_INFO_REQUEST(142),
- USER_INFO(143),
- USER_NAME_SET(144),
- USER_OPTIONS_SET(145),
- USER_PROG_REQUEST(146),
- USER_PROG_INFO(147),
- USER_PROG_SET(148),
- AUTO_RELOCK_PROG_REQUEST(149),

- AUTO_RELOCK_PROG_INFO(150),
- AUTO_RELOCK_PROG_SET(151),
- LOG_REQUEST(152),
- LOG_INFO(153),
- KEY_BLE_APPLICATION_BOOTLOADER_CALL(154),
- DAYLIGHT_SAVING_TIME_OPTIONS_REQUEST(155),
- DAYLIGHT_SAVING_TIME_OPTIONS_INFO(156),
- DAYLIGHT_SAVING_TIME_OPTIONS_SET(157),
- FACTORY_RESET(158);

Connection/session establishment

After the connection is established on the lower bluetooth layer, CONNECTION_REQUEST and CONNECTION_INFO packets are exchanged. If the app isn't paired to the device yet, it requests with userNumber = 0xff, and the lock answers with the userNumber set to the next free user number. Otherwise, the app puts the userNumber it got told at pairing time. This allows the lock to know, which pairingKey to use for this session. Randomly generated nonces are exchanged to prevent full session replay attacks. Each peer generates and sends a nonce, which must be used by the other side when encrypting and authenticating packets for that peer. The packetCounters for both directions, which are used to prevent replay attacks during the session, are initialized to zero.

Secure packets

For secure packets, the command type byte is transferred unencrypted as the first byte, just as with plain text packets. After that, encrypted data (whose length depends on the command type), a packet counter, and a 4 byte authentication value (HMAC-like) follows.

```
struct secure_packet {
        uint8_t commandType;
        char encryptedData[n];
        uint16_t packetCounter;    // number of the packet in this connection (per direction)
        uint32_t authenticationValue;
}
```

The plaintext data is padded with zeroes before encryption such that the length of the resulting secure_packet is a multiple of 15, therefore filling up transport fragments completely.

Encryption

For encryption, a custom variation of AES counter mode is used. Payloads are split in blocks of 16 byte length. For each block a 16 byte block_nonce is built (see below), which is AES-encrypted with the pairingKey and used as an intermediate key. The block is then XORed with the intermediate key to produce the encrypted output. For decryption, the same process can be repeated, because two times XOR with the same key produces the original value.

```
struct block_nonce {
        uint8_t nonceType;  // 1 for encryption, 9 for authentication
```

```
        uint8_t commandType;
        char directionalNonce[8]; //either deviceNonce or applicationNonce, depending on direction
        uint16_t zero;                //always zero
        uint16_t packetCounter;   // number of the packet in this connection (per direction)
        uint16_t blockCounter;    // number of block in the packet
}
```

Authentication

To calculate the authentication value, the following steps are taken: The payload is zero-padded to a multiple of 16 bytes, and split in blocks of 16 bytes. A nonce for the whole hash is built as described above (block_nonce), but with nonceType = 9 and blockCounter = len(unpadded payload). It is encrypted with the pairingKey and stored as preHash. For each block, the preHash is XORed with the block, the result is encrypted with the pairingKey and stored as the new preHash. Another nonce is built with nonceType = 9 and blockCounter = 0, encrypted with the pairingKey and XORed with the preHash. The first four bytes of the result is the resulting hash.

Pairing

In the above description of the encrypted packets, a pairingKey is referenced. The 16 byte pairingKey is randomly generated by the Android app during the pairing process, and transferred to the lock.

Pairing process (user perspective)

The user clicks on the pairing button in the app and is then guided through the following steps: With the lock, two identical key cards are shipped, containing a QR code, which the user has to scan first. Then the app advises to put the lock into pairing mode, by holding the physical unlock button for a few seconds. Afterwards the app establishes a bluetooth connection and pairs with the lock. The user can then lock, unlock and open the door via buttons in the app.

Internal view

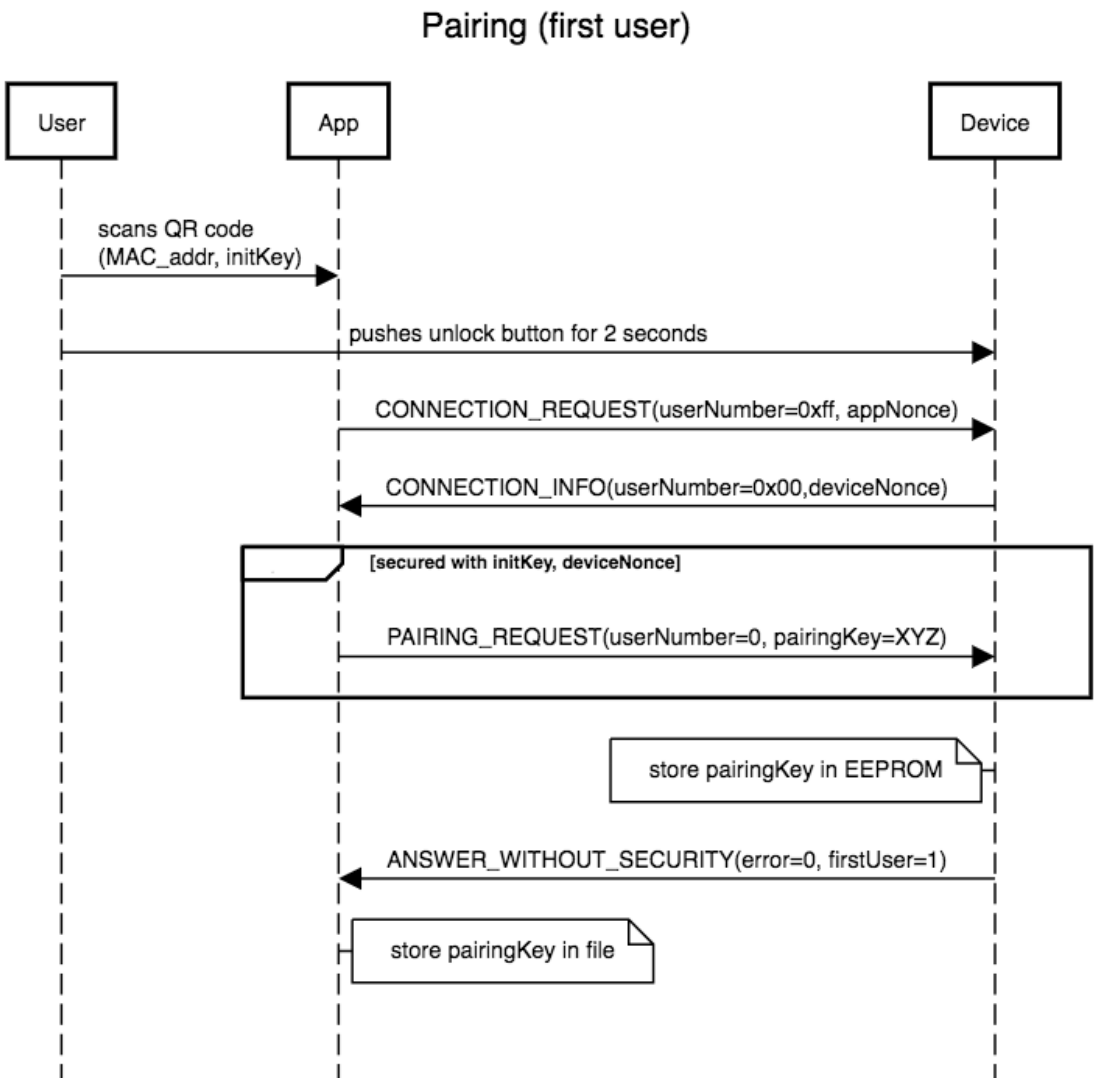

Illustration 5: Sequence diagram of pairing process

The QR code on the card contains the lock‘s bluetooth MAC address, its serial number, and a secret initialization key (which is also programmed into the STM8 during production of the device). Therefore the app can directly connect to the correct bluetooth device and send a CONNECTION_REQUEST, answered by CONNECTION_INFO. Pushing the button puts the lock in a special pairing mode, allowing the PAIRING_REQUEST command to be used, which the app will send to the device. It contains the randomly generated pairingKey. The PAIRING_REQUEST is encrypted and authenticated similar to other secure packets, but instead of the pairingKey, the initialization key is used for encrypting the nonces.

The lock answers the PAIRING_REQUEST with an ANSWER_WITHOUT_SECURITY packet. If the lock was indeed in pairing mode, and deems the authentication value of the PAIRING_REQUEST correct, the error flag is unset. If no other users are paired yet, the isFirstUser flag is set as well.

Security analysis

To pair a phone, the user needs to push a physical button, therefore an adversary who has the QR code, but is outside the door, can‘t pair to the lock. This prevents an attack where somebody who previously owned the lock, can connect to it later on from outside. It is also possible to disable the pairing completely from inside the app, preventing an attack where somebody who is temporarily inside the house, and found the QR code, can pair to the lock.

Replay of a pairing without knowledge of the initialization key would be impossible due to the nonces exchanged in connection establishment, and also useless because the pairingKey can’t be decrypted.

Replay attacks of unlock commands sent by legitimate users are also prevented by the combination of nonces exchanged in the connection handshake and packet counters during connection.

A possible attack would be to sell a lock where the attackers phone is already paired to the device, hoping that the new user doesn’t notice. In the app, a list of all paired phones can be viewed, however an unsuspecting user might not open it, as it is hidden in a multilevel menu. (TODO: check whether a warning is displayed on pairing, if someone else is already paired)