Protocol Overview

This is mostly based off of reading the slightly deobfuscated source code of the Pax WebApp and Bluetooth packet logs.

All messages for controlling the device seem to go through a single service with UUID `8E320200-64D2-11E6-BDF4-0800200C9A66`. Of interest is a read characteristic with UUID `8E320201-64D2-11E6-BDF4-0800200C9A66` and write characteristic `8E320202-64D2-11E6-BDF4-0800200C9A66`; there is also a notification characteristic `8E320203-64D2-11E6-BDF4-0800200C9A66`. All commands are at least 16 bytes, but some commands can be longer. No padding is applied otherwise, as AES is used as if it were a stream cipher here. The first byte of each message is some sort of type indicator, which defines the format of the rest of the message's contents.

Blob data sent to/from the device appears to be AES-128 encrypted, in OFB mode. Every packet that is sent/received resets the cipher state, however, so there's no actual cipher state needed to be maintained between messages. The key for the cipher is derived during initialization from the device serial. All messages have a 16 byte IV at the end that is used for the OFB cipher to decode that message.

See this Python code for an example of how the encryption/decryption of packets is implemented.

There's also a more complete example on actually connecting to, and interfacing with a real device.

Note: This encryption mechanism only applies to the Pax Era and the Pax 3; the new Pax Era Pro has a more sophisticated protocol using nonces and AES-CTR mode, as well as a connection handshake that is not covered here.

Notifications

The notification characteristic is used relatively widely to indicate that new data is available. It appears to always send a single byte, which is the first byte of the encrypted packet that is ready to be sent.

Message Structure

Messages are read in blocks of at least 32 bytes (?). This is treated as two separate 16-byte values: the first is a standard device to host attribute message, while the latter is the 16-byte IV to use for

cipher operations with this message. In all cases, the device key is used.

Device Key Derivation

The key used for encryption is derived by concatenating the serial number string (this is always 8 characters long) to itself to form a 16 byte blob. It is then encrypted using ECB with a fixed key (`F7 C8 66 C3 8F 78 75 30 86 29 3B D5 7D D3 25 40`), and the first 16 bytes are stored as the key with which all further messages are encrypted.

Message Types

Regardless of contents of the rest of the message, the first byte of the message is always reserved to indicate its type to allow correct decoding. Messages are simply packed structs; no special encoding is required. All multibyte integers are stored in little endian byte order. See here for a list of all message types.

Known message types are as follows:

LED brightness

Message type: `ATTRIBUTE_BRIGHTNESS`

```
struct {
    // 0 = 0%, 0x7F = 100%
    uint8_t brightness;
    // reserved (0) for Era
    uint8_t command;
};
```

Lock state

Message type: `ATTRIBUTE_LOCKED`

```
struct {
    // 0 = unlocked, 1 = locked
    uint8_t isLocked;
};
```

Device name

Message type: `ATTRIBUTE_DEVICE_NAME`

```
struct {
    // length of the string, in bytes
    uint8_t length;
    // name string. this is NOT null terminated
    char name[];
};
```

Heater set point

Message type: `ATTRIBUTE_HEATER_SET_POINT`

```
struct {
    uint16_t temp;
};
```

The temperature value is multiplied by 10; so 420°C is encoded as 4200 or 0x1068.

Status update

Message type: `ATTRIBUTE_STATUS_UPDATE`

This message, when sent to the device, triggers the transmission of all attribute values whose bits are set. For example, to read both `ATTRIBUTE_ACTUAL_TEMP` (1) and `ATTRIBUTE_CHARGE_STATUS` (7) simultaneously, the types value would be set to `0b10000001`.

```
struct {
    // bitmask of all attribute types to read out
    uint64_t types;
};
```

Revision #4
Created 4 October 2021 18:42:28
Updated 5 February 2022 23:46:14