Embedded Software Specification

High-Level Specification

V0.0.3

Revision	Date	Author	Notes
0.0.2	5/28/2022	Maurice McCabe	Add addendum for additional lock feature
			Clarify security model and cleanup swim lanes
0.0.3	5/29/2022	Maurice McCabe	Layer architecture. Buzzer command

Introduction

The scope of this specification is limited to the commands that are sent between the mobile app and the box. It is targeted to the app developer and the firmware developer. The intention is to specify the commands that need to be implemented to allow the box to perform as expected.

It is anticipated that the specification and subsequent implementation of commands in the firmware will provide adequate input to the hardware designer to identify the features required to support the firmware. Specifics of the hardware are beyond the scope of this specification.

The entire system we are designing depends on a server that maintains a database and is used to exchange messages between mobile devices running the app. One critical requirement of the system is that our messaging be encrypted at all times.

Therefore, the commands sent to the box are over a custom secure channel. This specification includes the portion of the security specification for secure messaging between the app and the box. This specification assumes that the connection between the app and the server has already been implemented. Details regarding the interface between the app and the server is beyond the scope of this specification.

Architecture

The integration of the App and the Box is implemented using three layers

1. Command Layer
   This is where the commands are implemented in the firmware on the Box. A corresponding API is implemented on the App to call the commands.
2. Security Layer
   This is responsible for maintaining a secure channel between the App and the Box
3. BLE Transport Layer
   This is a standard BLE connection between the App and the Box

BLE Transport Layer

This is where the connection between the App and the Box is established using standard BLE protocols. It is managed by the Security Layer. In our case it must provide support for receiving a Pair/Bonding token to establish a previously paired/bonded connection.

Security Layer

This is managed by the Command Layer and manages the BLE Transport Layer. It guarantees a secure connection over the BLE Transport Layer

Security Components

These are the security components used throughout the rest of this document.

Notations

- **Sign(privateKey, data)**
  Signing data using a private key, the data is arbitrary and can also be a public key.
  Returns encrypted data
- **Private[data]**
  Container for data signed using a private key, ie, encrypted data
  This is the only way that data appears on a secure channel
  The data has to be serialized before encrypting and deserialized after unencrypted.
  We are using Protocol Buffers to serialize/deserialize
- **Verify(publicKey, Private[data])**
  Verifying signed data using a public key
  Returns unencrypted data

Actors

- **Owner**: The car owner
- **Third Party User (TPU)**: The user that needs temporary access to the car
- **Server**: The authentication server responsible for managing all user credentials and verifications
- **Box**: The box in the car

Security Layer Initialization

Base Keys

All parties will generate Asymmetric cryptographic keys

- Server Keys: Spub, Sprv
- Owner Keys: Opub, Oprv
- Box Keys: Bpub, Bprv
- Third Party User Keys: Upub, Uprv

Assumptions

- The Owner and Third Party Users (TPU) have registered on the Server and have exchanged public keys with the Server
- The Server has signed and returned the provided public keys using
  - Sign(Sprv, Oprv) and
  - Sign(Sprv, Uprv).

  Using the provided notation this means that the data returned to the Owner and TPU is respectively
  - Private[Sprv(Opub)]
  - Private[Sprv(Upub)]

  This data has been verified by the Owner and TPU using
  - Verify(Opub, Private[Sprv(Opub)]) and
  - Verify(Upub, Private[Sprv(Upub)])

Command Layer

This is the layer used by the application developer. It is responsible for establishing the Security Layer and sending commands from the Box to the App. It is dependent on the Security Layer and the BLE Transport Layer. The command layer appears to the app developer as an API that supports the required commands. Other than initializing the dependent layers, the app developer only uses the API to implement the App. On the Box side the firmware developer implements the commands.

The command layer is implemented using Protocol Buffers which is a tool that converts requests/responses, i.e. commands, between the App and the Box to data that can be encrypted/unencrypted. Only encrypted data can appear on a secure channel.
To send commands from the App to the Box we use three layers

1. Command Layer
   Commands are serialized/deserialized into data using ProtoBufs
2. Security Layer
   Data is encrypted/decrypted using Asymmetric Public/Private Key Encryption
3. BLE Transport Layer
   Encrypted data is transported between App and Box. This is a secure channel.

On the App side:

1. The commands are serialized using ProtoBufs and sent to the security layer as data.
2. In the security layer the data is encrypted and sent to the BLE channel.
3. In the BLE transport layer the encrypted data is sent to the Box

On the Box side:

1. An encrypted message is received from the BLE layer by the security layer, unencrypted and sent to the command layer for processing.

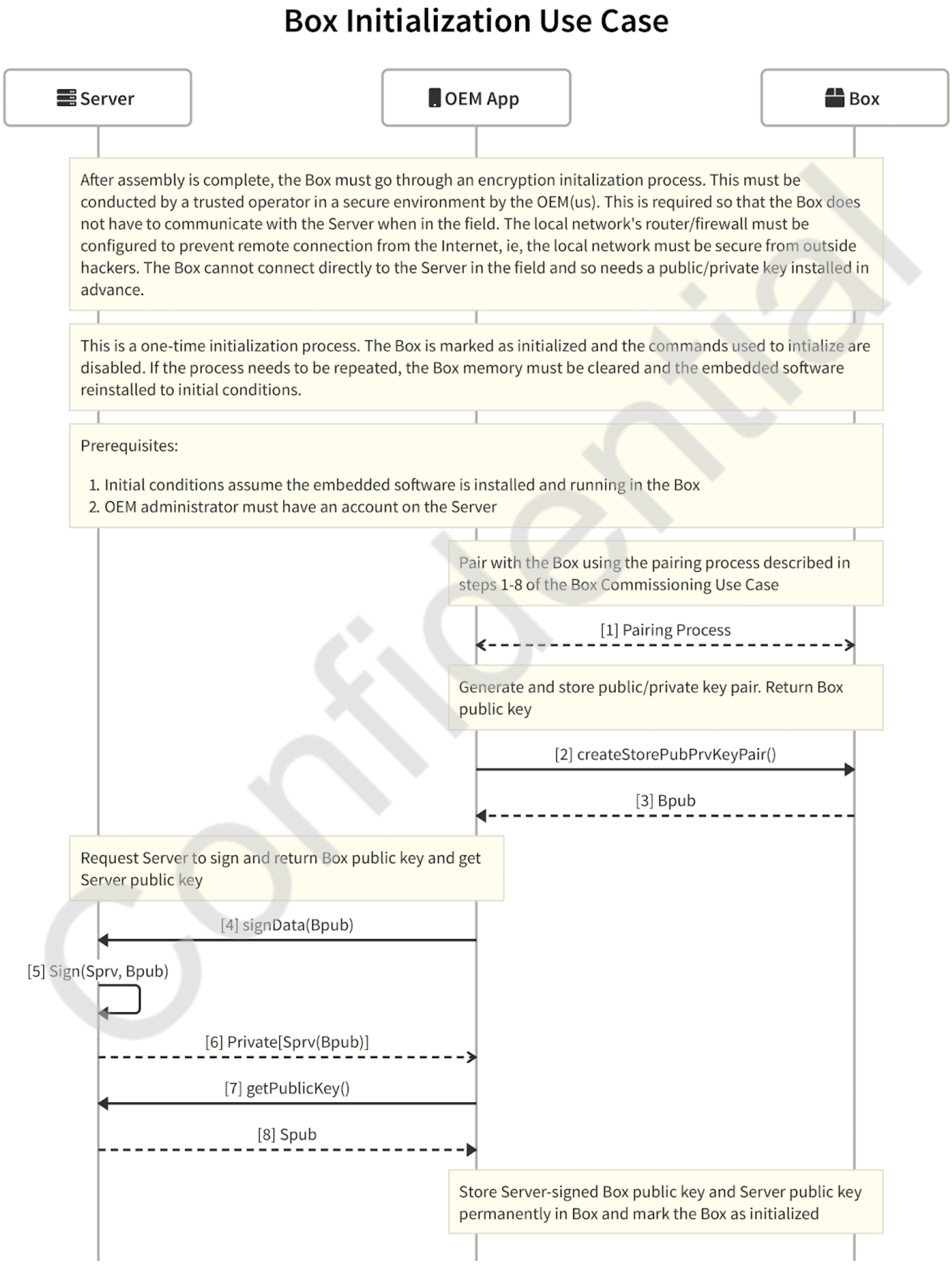
Responses from the Box are sent to the App using the same three layers.

Protocol Buffers implement the command layer via a configuration file that defines the supported commands. Code is auto-generated for the App side and the Box side separately. The data produced is integrated to the security layer.

Box-focused Use Cases

The following use cases are specific to the commands sent between the app and the box. We are using a swim lane diagram format to illustrate the division of responsibilities between the app and the box required to implement the use cases.

Box Initialization Use Case



Box Initialization Use Case

After assembly is complete, the Box must go through an encryption initalization process. This must be conducted by a trusted operator in a secure environment by the OEM(us). This is required so that the Box does not have to communicate with the Server when in the field. The local network's router/firewall must be configured to prevent remote connection from the Internet, ie, the local network must be secure from outside hackers. The Box cannot connect directly to the Server in the field and so needs a public/private key installed in advance.

This is a one-time initialization process. The Box is marked as initialized and the commands used to intialize are disabled. If the process needs to be repeated, the Box memory must be cleared and the embedded software reinstalled to initial conditions.

Prerequisites:

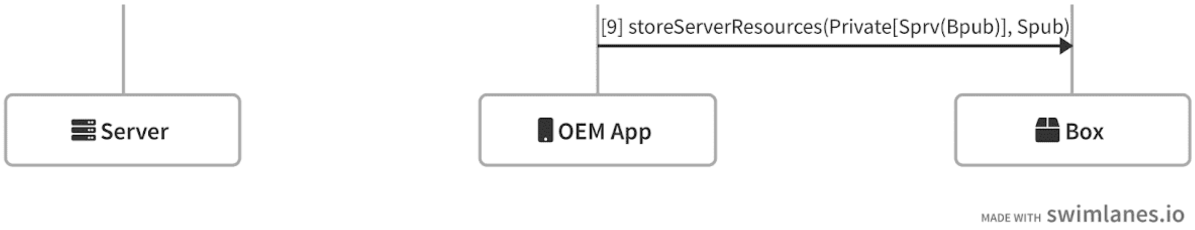
1. Initial conditions assume the embedded software is installed and running in the Box
2. OEM administrator must have an account on the Server

Pair with the Box using the pairing process described in steps 1-8 of the Box Commissioning Use Case

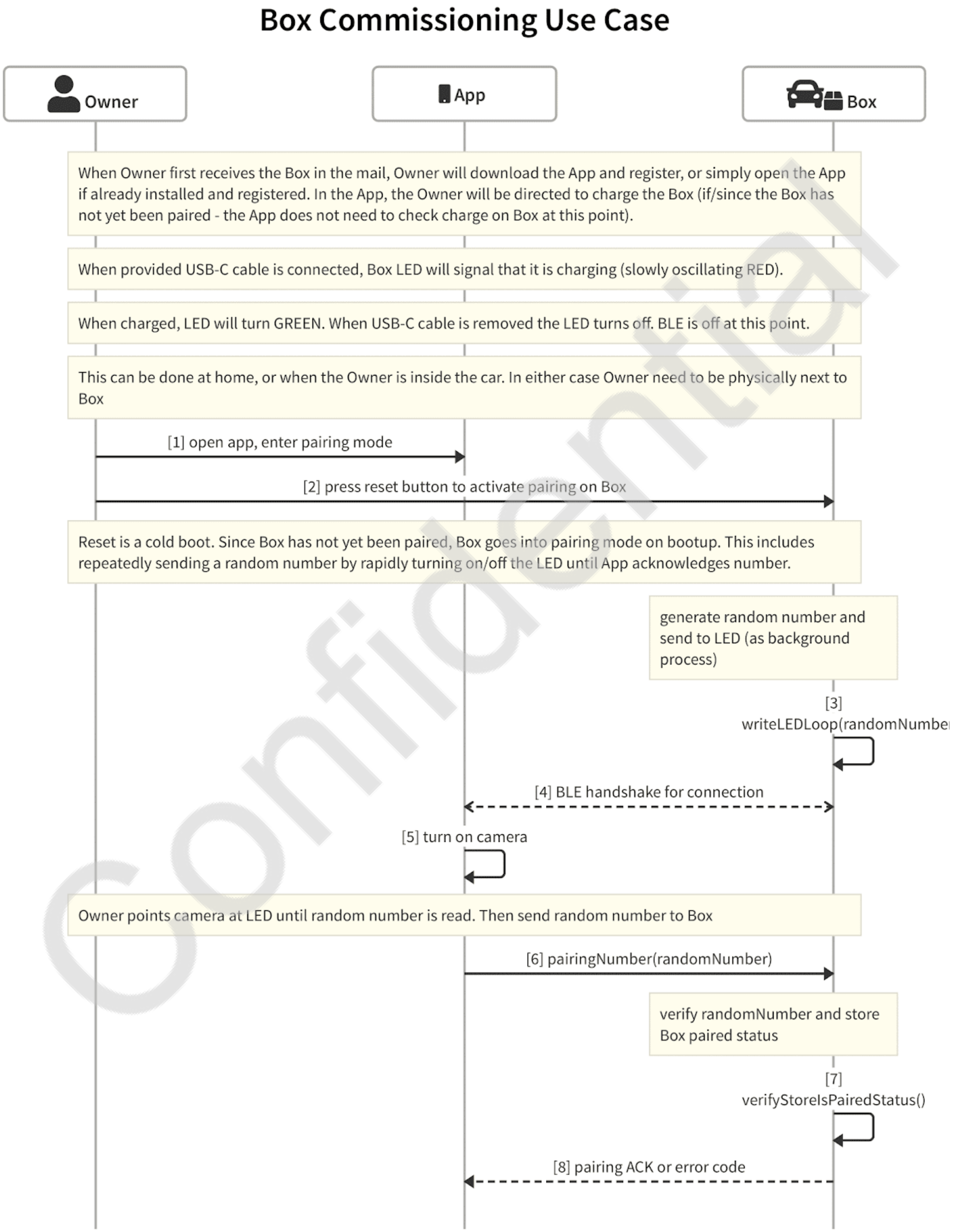
Generate and store public/private key pair. Return Box public key

Request Server to sign and return Box public key and get Server public key

Store Server-signed Box public key and Server public key permanently in Box and mark the Box as initialized


[9] storeServerResources(Private[Sprv(Bpub)], Spub)
Server
OEM App
Box
MADE WITH swimlanes.io

Box Commissioning Use Case



Box Commissioning Use Case

When Owner first receives the Box in the mail, Owner will download the App and register, or simply open the App if already installed and registered. In the App, the Owner will be directed to charge the Box (if/since the Box has not yet been paired - the App does not need to check charge on Box at this point).

When provided USB-C cable is connected, Box LED will signal that it is charging (slowly oscillating RED).

When charged, LED will turn GREEN. When USB-C cable is removed the LED turns off. BLE is off at this point.

This can be done at home, or when the Owner is inside the car. In either case Owner need to be physically next to Box

[1] open app, enter pairing mode

[2] press reset button to activate pairing on Box

Reset is a cold boot. Since Box has not yet been paired, Box goes into pairing mode on bootup. This includes repeatedly sending a random number by rapidly turning on/off the LED until App acknowledges number.

generate random number and send to LED (as background process)

[3] writeLEDLoop(randomNumbe

[4] BLE handshake for connection

[5] turn on camera

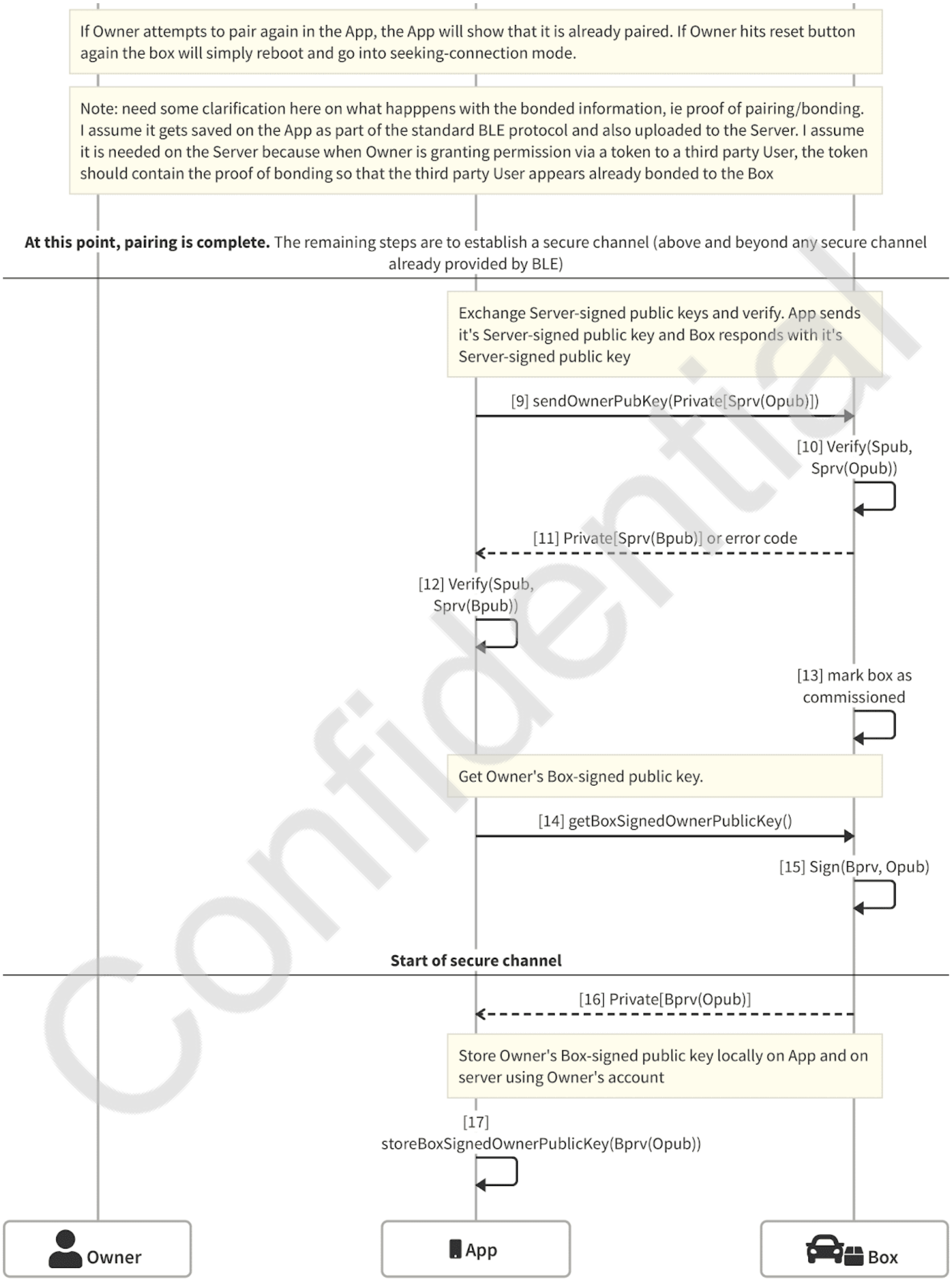
Owner points camera at LED until random number is read. Then send random number to Box

[6] pairingNumber(randomNumber)

verify randomNumber and store Box paired status

[7] verifyStoreIsPairedStatus()

[8] pairing ACK or error code



If Owner attempts to pair again in the App, the App will show that it is already paired. If Owner hits reset button again the box will simply reboot and go into seeking-connection mode.

Note: need some clarification here on what happpens with the bonded information, ie proof of pairing/bonding. I assume it gets saved on the App as part of the standard BLE protocol and also uploaded to the Server. I assume it is needed on the Server because when Owner is granting permission via a token to a third party User, the token should contain the proof of bonding so that the third party User appears already bonded to the Box

At this point, pairing is complete. The remaining steps are to establish a secure channel (above and beyond any secure channel already provided by BLE)

Exchange Server-signed public keys and verify. App sends it's Server-signed public key and Box responds with it's Server-signed public key

[9] sendOwnerPubKey(Private[Sprv(Opub)])

[10] Verify(Spub, Sprv(Opub))

[11] Private[Sprv(Bpub)] or error code

[12] Verify(Spub, Sprv(Bpub))

[13] mark box as commissioned

Get Owner's Box-signed public key.

[14] getBoxSignedOwnerPublicKey()

[15] Sign(Bprv, Opub)

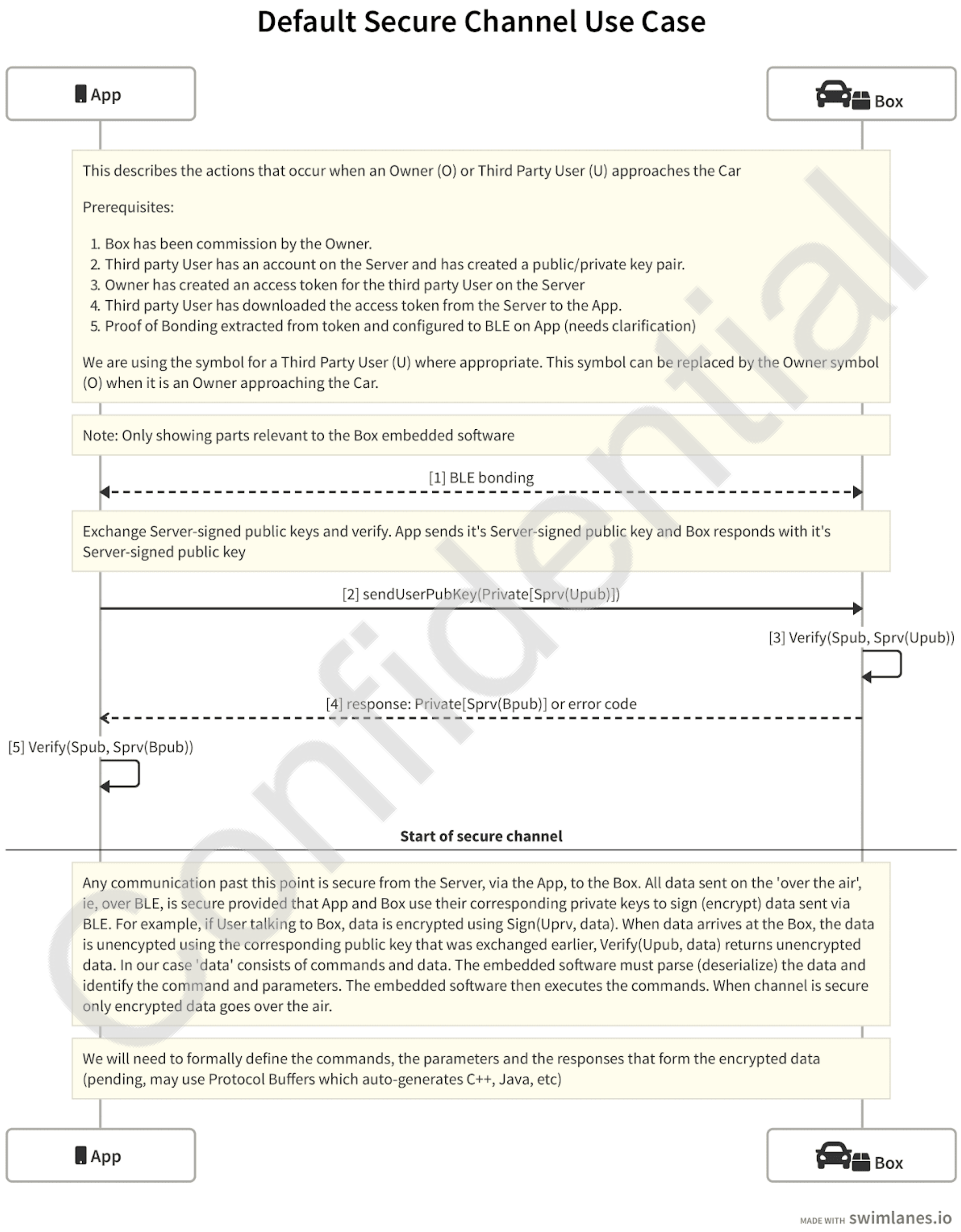
Start of secure channel

[16] Private[Bprv(Opub)]

Store Owner's Box-signed public key locally on App and on server using Owner's account

[17] storeBoxSignedOwnerPublicKey(Bprv(Opub))

Default Secure Channel Use Case



Default Secure Channel Use Case

This describes the actions that occur when an Owner (O) or Third Party User (U) approaches the Car

Prerequisites:

1. Box has been commission by the Owner.
2. Third party User has an account on the Server and has created a public/private key pair.
3. Owner has created an access token for the third party User on the Server
4. Third party User has downloaded the access token from the Server to the App.
5. Proof of Bonding extracted from token and configured to BLE on App (needs clarification)

We are using the symbol for a Third Party User (U) where appropriate. This symbol can be replaced by the Owner symbol (O) when it is an Owner approaching the Car.

Note: Only showing parts relevant to the Box embedded software

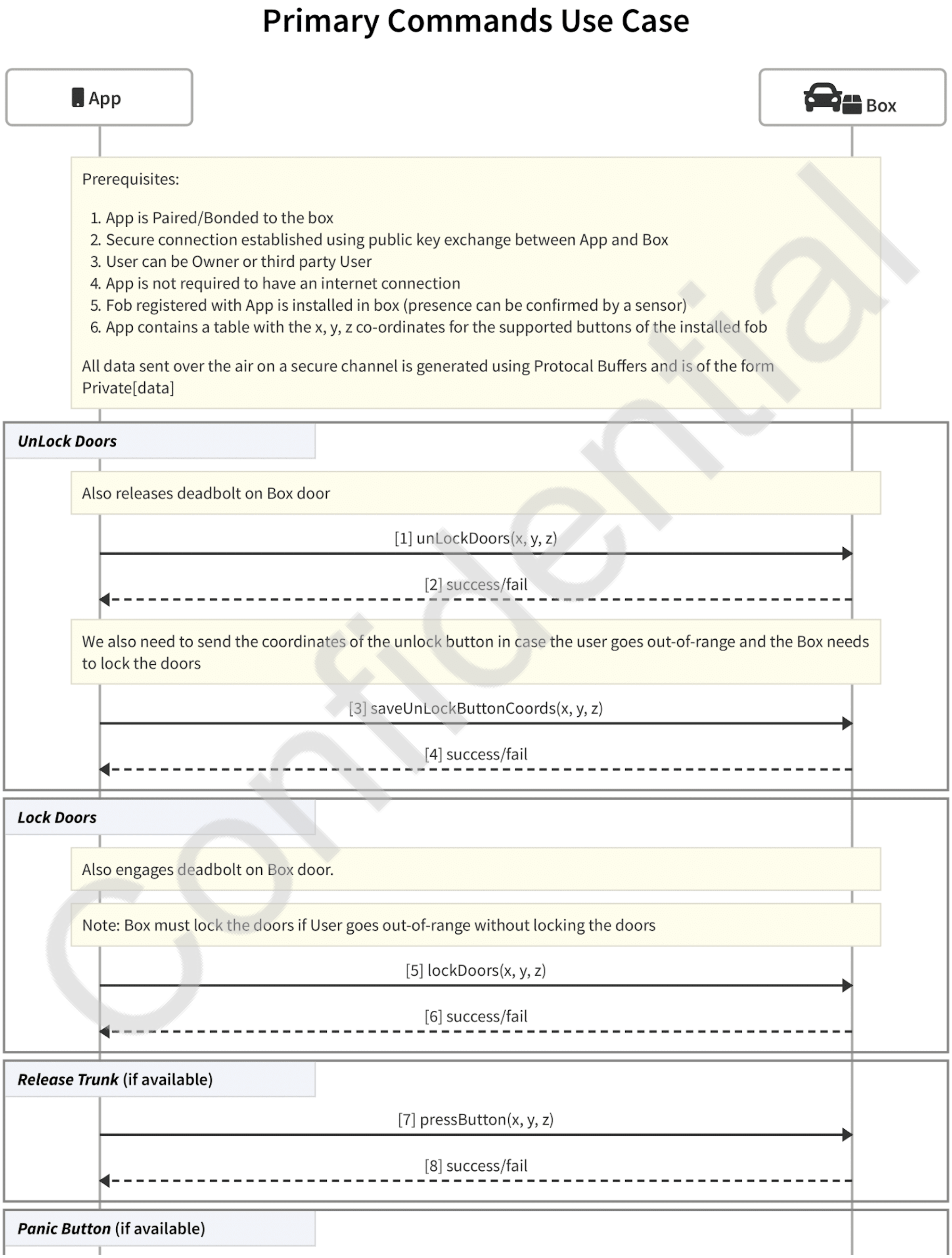
Exchange Server-signed public keys and verify. App sends it's Server-signed public key and Box responds with it's Server-signed public key

[2] sendUserPubKey(Private[Sprv(Upub)])
[3] Verify(Spub, Sprv(Upub))
[4] response: Private[Sprv(Bpub)] or error code
[5] Verify(Spub, Sprv(Bpub))
Start of secure channel

Any communication past this point is secure from the Server, via the App, to the Box. All data sent on the 'over the air', ie, over BLE, is secure provided that App and Box use their corresponding private keys to sign (encrypt) data sent via BLE. For example, if User talking to Box, data is encrypted using Sign(Uprv, data). When data arrives at the Box, the data is unencypted using the corresponding public key that was exchanged earlier, Verify(Upub, data) returns unencrypted data. In our case 'data' consists of commands and data. The embedded software must parse (deserialize) the data and identify the command and parameters. The embedded software then executes the commands. When channel is secure only encrypted data goes over the air.

We will need to formally define the commands, the parameters and the responses that form the encrypted data (pending, may use Protocol Buffers which auto-generates C++, Java, etc)

Primary Commands Use Case



Primary Commands Use Case

Prerequisites:

1. App is Paired/Bonded to the box
2. Secure connection established using public key exchange between App and Box
3. User can be Owner or third party User
4. App is not required to have an internet connection
5. Fob registered with App is installed in box (presence can be confirmed by a sensor)
6. App contains a table with the x, y, z co-ordinates for the supported buttons of the installed fob

All data sent over the air on a secure channel is generated using Protocal Buffers and is of the form Private[data]

UnLock Doors

Also releases deadbolt on Box door

[1] unLockDoors(x, y, z)

[2] success/fail

We also need to send the coordinates of the unlock button in case the user goes out-of-range and the Box needs to lock the doors

[3] saveUnLockButtonCoords(x, y, z)

[4] success/fail

Lock Doors

Also engages deadbolt on Box door.

Note: Box must lock the doors if User goes out-of-range without locking the doors

[5] lockDoors(x, y, z)

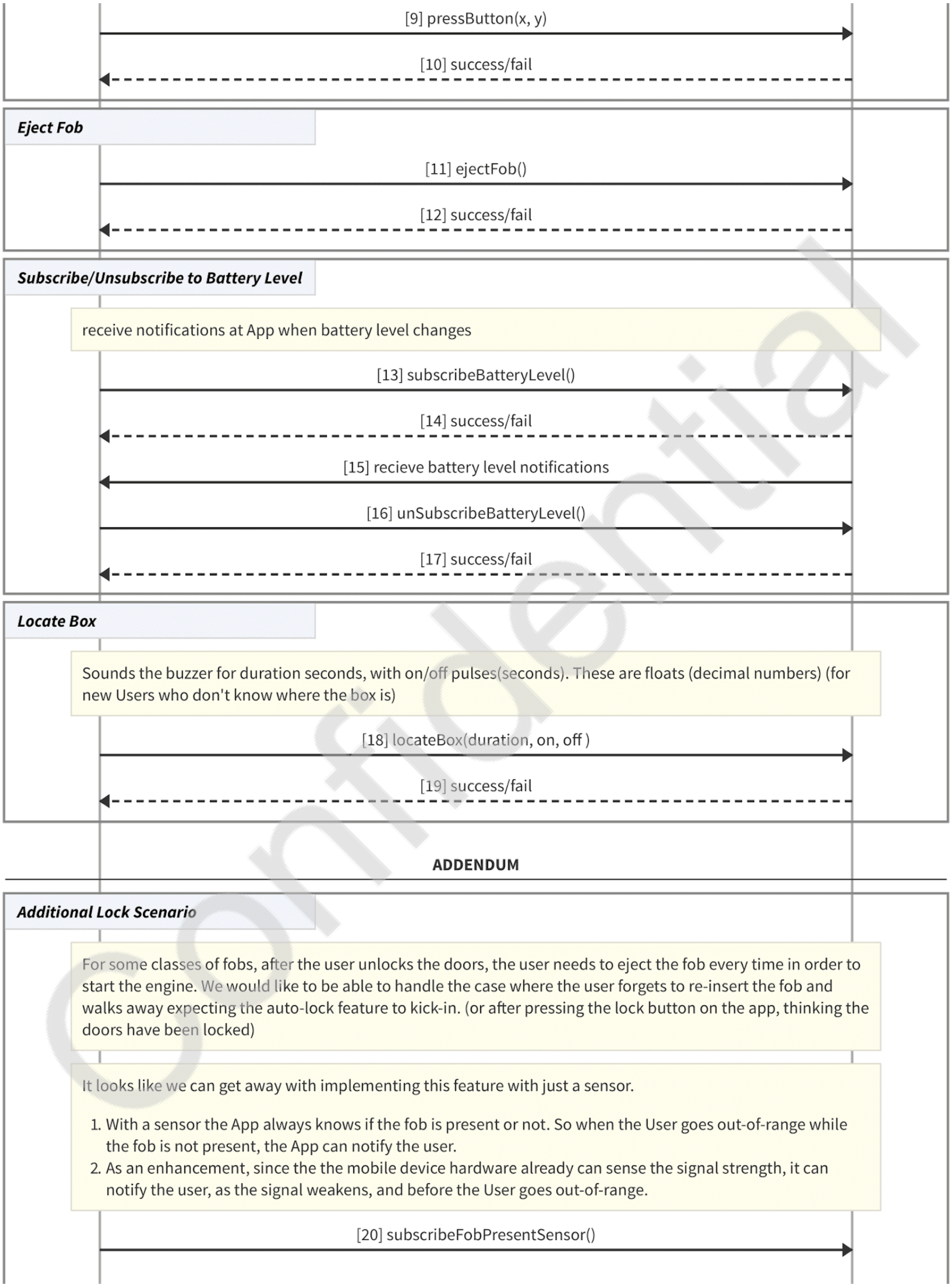
[6] success/fail

Release Trunk (if available)

[7] pressButton(x, y, z)

[8] success/fail

Panic Button (if available)


[9] pressButton(x, y)
[10] success/fail
Eject Fob
[11] ejectFob()
[12] success/fail
Subscribe/Unsubscribe to Battery Level
[13] subscribeBatteryLevel()
[14] success/fail
[15] recieve battery level notifications
[16] unSubscribeBatteryLevel()
[17] success/fail
Locate Box
[18] locateBox(duration, on, off)
[19] success/fail
ADDENDUM
Additional Lock Scenario
[20] subscribeFobPresentSensor()

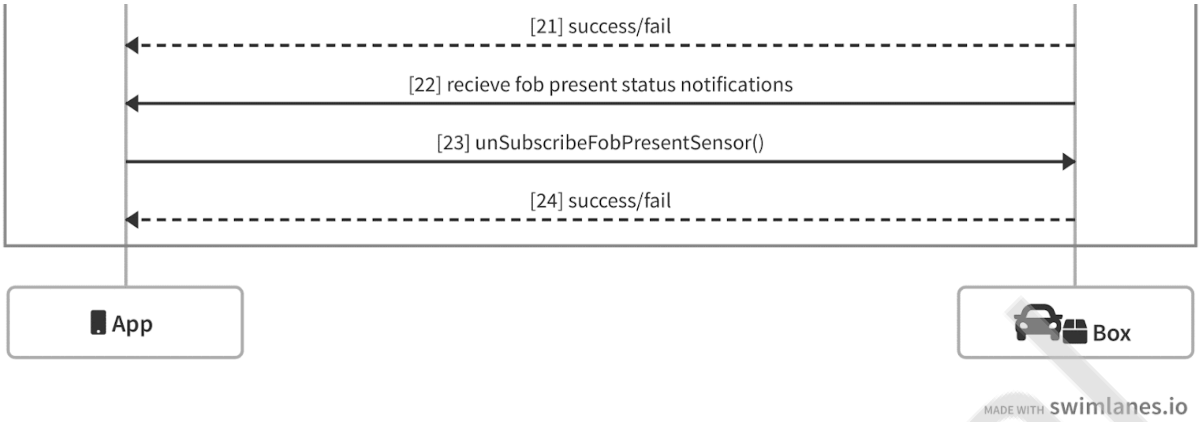
receive notifications at App when battery level changes

Sounds the buzzer for duration seconds, with on/off pulses(seconds). These are floats (decimal numbers) (for new Users who don't know where the box is)

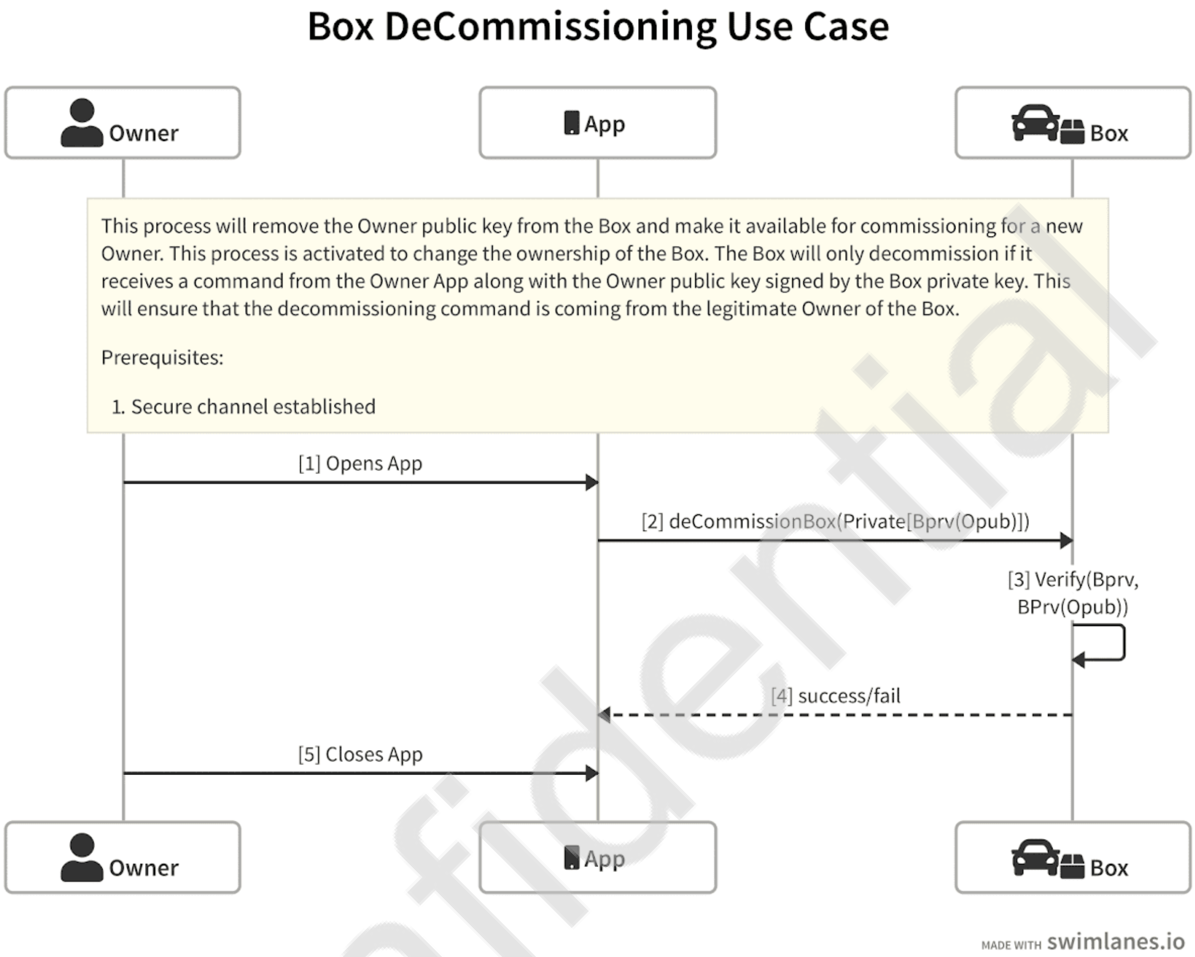
For some classes of fobs, after the user unlocks the doors, the user needs to eject the fob every time in order to start the engine. We would like to be able to handle the case where the user forgets to re-insert the fob and walks away expecting the auto-lock feature to kick-in. (or after pressing the lock button on the app, thinking the doors have been locked)

It looks like we can get away with implementing this feature with just a sensor.

1. With a sensor the App always knows if the fob is present or not. So when the User goes out-of-range while the fob is not present, the App can notify the user.
2. As an enhancement, since the the mobile device hardware already can sense the signal strength, it can notify the user, as the signal weakens, and before the User goes out-of-range.


[21] success/fail
[22] recieve fob present status notifications
[23] unSubscribeFobPresentSensor()
[24] success/fail
App
Box
MADE WITH swimlanes.io

Box DeCommissioning Use Case



Box DeCommissioning Use Case

This process will remove the Owner public key from the Box and make it available for commissioning for a new Owner. This process is activated to change the ownership of the Box. The Box will only decommission if it receives a command from the Owner App along with the Owner public key signed by the Box private key. This will ensure that the decommissioning command is coming from the legitimate Owner of the Box.

Prerequisites:

1. Secure channel established

Implementation Support

Implementation of the Command Layer and Security Layer in the App and Box will be co-ordinated/implemented by Hassaan and Maurice including Sign and Verify and management of the public and private keys.

The remainder of the firmware implementation, including the BLE Transport Layer, and the actual commands, will be implemented by Khandoker with support from Don with the hardware, peripherals, schematics.

Integration with the app will be supported by Maurice