eCTF 2023
DESIGN DOCUMENT

Version 1.0

Singapore Management University

Overview **1**
Implementation **2**
Space configurations 2
Security Requirements 2
Confidentiality 2
Integrity and Authenticity 2
Secure Fob Pairing 2
Security Details 2
Encryption and Decryption Subroutine (Build Deployment) 2
Functional Requirements 4
Build car and Paired Fob 4
Build Unpaired Fob 5
Pair Fob 5
Package Feature 6
Enable Feature 7
Unlock car 7

Overview

The Protected Automotive Remote Entry Device (PARED) system contains both the host-tools as well as the various firmwares for the car and the fobs - both paired and unpaired. This document would outline the details on the high level implementation of these components, focusing on the security details. It would pen down information about the type of keys used for encryption and integrity checks, the location of key storage, the flow of the protocols used to ensure for secure communication between the devices and lastly, other techniques to add security.

Implementation

Space configurations

To accommodate the space and size requirements, we have decided to structure the flash and EEPROM layout similar to the image shown in the Technical Specifications.

For cryptographic keys and sensitive information, they would be **stored (hard-coded) into the firmware binary**. This is to simplify development and to avoid overcomplicating the various security operations. Hence, the EEPROM (as of the time of writing) would only be used to store the various messages in the last 256 bytes, leaving the rest of the memory unused.

Security Requirements

Confidentiality

To protect the confidentiality of the information, the Monocypher library (as well as its Python binding pymonocypher) for Authenticated Encryption. This mechanism ensures both confidentiality and integrity of the messages and firmware packages.

The library uses state of the art XChaCha20 and Poly1305 (RFC8439), both of which are relatively secure as of the time of writing.

Integrity and Authenticity

To further ensure the integrity of the packaged feature, signatures and shared symmetric keys would be used for encryption, decryption and authenticated key exchange.

While the high level cryptography operations specified do not follow any well known documented standards, we believe that it has forward secrecy to some extent.

Secure Fob Pairing

Authenticated Key Exchange mechanisms would be put in place for pairing of devices. This meant that there would be an implicit trusted third party (TTP), which we would term it the

Certificate Authority (CA), who would be in charge of signing the public keys used in the various car and fob firmwares. This would mean that the chance of impersonation and Man-in-the-Middle (MitM) attacks would be reduced.

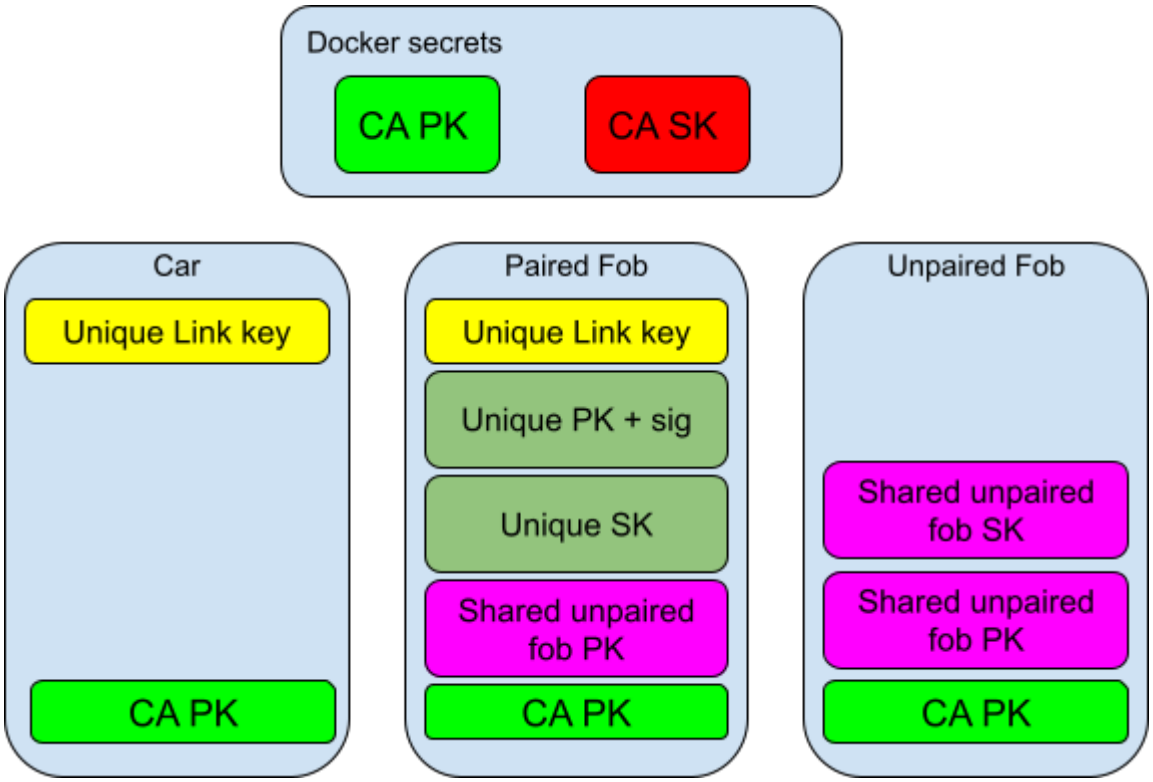
Security Details

Encryption and Decryption Subroutine (Build Deployment)

The PARED system would adopt the use of multiple keys for multiple purposes, to avoid the total loss of security once a single key is obtained.

In detail, the build deployment stage would create a few high entropy keys by pulling data from /dev/urandom:

1. **Deployment (CA) public and secret key**: The secret key would be used to sign the feature packs and the public key of every paired fob - which is unique. Evidently, the secret key should be kept highly secret and in the host secrets docker volume only. On the other hand, the public key of the CA will be hardcoded in every device for verification of the different critical components.
2. **Car-Fob link key**: **Unique key that is generated during the car-fob build process** and would be hardcoded into the relevant car and paired fob flash memory. Each car-fob pair would have a different link key. This key is mainly used for the unlocking of cars and other secure communications between the paired fob and car.
3. **Paired fob pk and sk**: Long term key pair used specifically for authenticated key exchange between unpaired fobs and paired fobs. Each key pair is signed by the CA and **each car-fob entity will have a unique key pair**.
4. **Unpaired fob pk and sk:** Long term key hardcoded into every single unpaired fob. The pk is distributed and available in all paired fobs. Similarly, the pk is signed by the CA.



Overview of keys storage and generation.

Functional Requirements

Our team has decided to keep the design simple and easy to understand. Complexity might make a system more robust but we believe that a simple system will be less prone to human and coding errors.

Build car and Paired Fob

During the build process, the host-tools would generate a unique high entropy (256 bit) key for each car-fob pair, also known as the *link-key*. This key would be used to encrypt the communication between the car and the fob. It would be hardcoded into the car's and fob's firmware.

Understandably, the owner of the car (car 0) would be able to view this key in an unencrypted manner but that will not compromise the security of the other cars, given that the firmware of other cars are encrypted.

Surely, debug access would be disabled when building the car and fob firmware to prevent adversaries from stepping through the code and retrieving the key easily during runtime.

The remaining secrets such as the paired fob public and private keys as well as their signatures would be generated during the building of the car and saved into the docker volume. This secret would be read and used during the building of the fob.

Build Unpaired Fob

All unpaired fobs would come with the same secret key (unpaired fob keypairs) used specifically for pairing purposes. Its sole purpose is to derive a fresh set of temporary keys during each pairing attempt to encrypt the communication and data sent between the paired and unpaired fob through the UART1 interface. The public key will be distributed to all paired fobs for future pairing purposes. Hence, this set of keys would be generated during the *deployment* phase, which is also in charge of generating the CA key pairs.

Pair Fob

During the pairing process, the paired fob (PF) will first verify that the pairing pin is correct in the first place. Else, it would just return early.

Next, once verified, it will move onto the second phase, by first sending the *SIG_PK_PACKET.* This packet contains the public key of the PF, the signature of that key (by the CA) and an ephemeral public key generated on the fly. After receiving the data, the unpaired fob (UPF) will then validate that the PF's public key is valid by checking that it is signed by the CA, using the hardcoded CA's public key. If the key is invalid, the process terminates.

Secondly, once the UPF is confident that the public key is valid, it would also generate an ephemeral key-pair and send its temporary public key back to the PF for authenticated key exchange.

Upon the derivation of the *shared_secret* using the classic Diffie-Hellman exchange, a longer set of shared keys would be derived by using Blake2b hashing. Specifically, it would be:

Key1 || Key2 = H(shared_secret || PF's temp pk || UPF's temp pk)

Once the session keys are established, which is supposed to be fresh every round, replay attacks and impersonation attacks are greatly reduced. Moreover, the scheme guarantees that there is some form of forward secrecy.

At this point in time, confidentiality of communications can be guaranteed using the fresh session keys. However, authentication has yet to occur. To prove one another's identity, their public key (CA signed) would be used for signing. Here are the details:

The PF will prove its identity by sending the UPF an encrypted, signed package.
It will craft a package in the form where E is encryption:

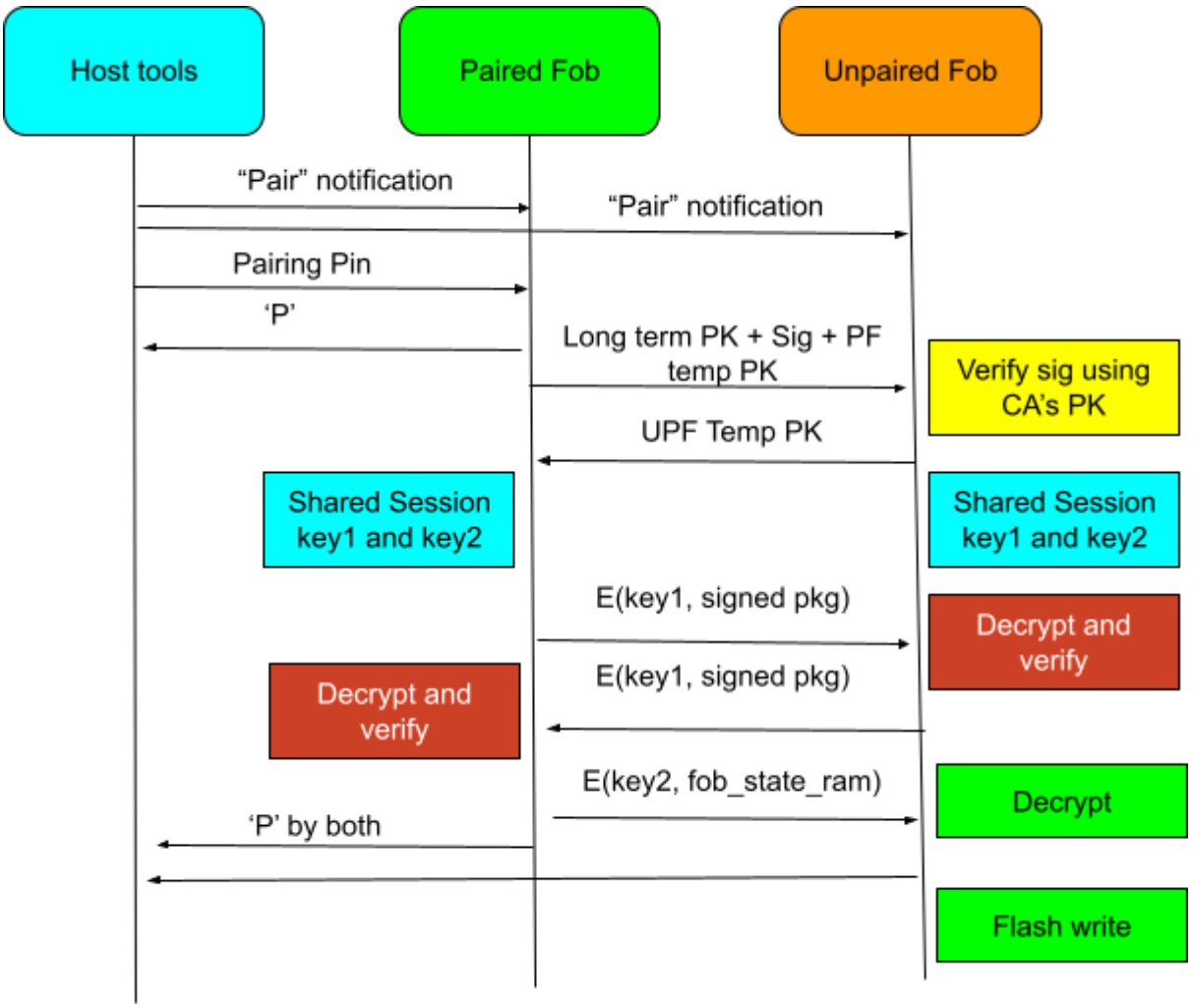
E_{key1}("PU" || $sign_{PF\_longterm\_SK}$(PF temp pk || UPF temp pk))

Once the UPF receives it, UPF has all the means to decrypt it using the fresh key1 and verify that the package is in the expected form, as well as verify the signature. Then, the process repeats in the opposite direction such that the encrypted, signed package is sent:

E_{key1}("UP" || $sign_{UPF\_longterm\_SK}$(UPF temp pk || PF temp pk))

Upon authentication, the paired fob will encrypt the data of its firmware and EEPROM to be sent to the unpaired fob using key2. Then the pairing process ends.

Lastly, should all checks succeed (performed by the encryption library), the unpaired fob will perform flash writes to modify its firmware into a paired fob.



Package Feature

In this step, the car_id-feature pair is simply encrypted using the car-fob link key. Hence, only when the attacker knows the secret link key, then it will be able to decrypt the content or craft its own features. Also included in the package is the CA signature on the (plaintext) metadata. Since it is not meant to hide any details (the metadata is easily guessable), the *MAC then encrypt* scheme is chosen for easy verification at later stages too (start car).

Enable Feature

Using the encrypted package, only the fob corresponding to this package can decrypt and load the feature into the fob. At no point in time is the package transmitted in plaintext across the serial bridge or UART0. This is because the decryption would fail if the receiving fob does not have the correct link-key to decrypt the content (verified using the MAC). Even if the attacker creates one's own package using a potentially leaked link key, the package cannot be enabled without a corresponding signature from the CA.

Of course, basic sanity checks would also be conducted to prevent bugs arising from multiple package loads even if those features are already enabled previously.

Unlock car

The car-fob link key would be used to perform secure communication. The car unlock process is akin to the challenge-response scheme. Once the SW0 button is pressed, the fob will initiate the pairing by sending over UART1 to the car the magic bytes of “Unlock”.

Next, the car would first request authentication from the fob by generating some random bytes **R** as its challenge and sending it to the fob. Not any challenge is valid as the fob will verify that the challenge string can be decrypted correctly using the shared link-key!

Next, the fob will be required to encrypt the message (“Unlock” || **R**) using the link key (k), sending it to the car for verification. Upon successful decryption and verification, the car will unlock and send back either a success or failure message. This message determines if the fob goes into the next phase of trying to start the car, upon a successful ACK.

Assuming the challenge-response is successful, the fob will send the feature info saved prior to the car encrypted using the same link-key. The data contains the car_id, number of features active, feature number and their respective signatures.

The car should then be able to decrypt the content and verify the signatures for all the features before starting the car with the appropriate features. This process is confirmed as it sends the EEPROM data across UART0. Another indicator is that the LED would turn from red to green, signifying a successful start after a reset.