HackieBird
MITRE eCTF 2023
Final Design Document

Raghav Agrawal, Kyla Coles, Wesley Flynn, Erin Freck, Christian Johnson, Jack Kolenbrander, Thomas Rydzewski, Raj Sahu

Table Of Contents

Overview **2**
Environment Setup **3**
Unlock and Start Car **6**
Security Features Addressed 6
Commands Run 7
Pair Fob **7**
Security Features Addressed 7
Commands Run 7
Package Feature **8**
Commands Run 9
Enable Feature **9**
Security Features Addressed 9
Commands Run 10

Overview

Architecture

- Build PARED System
    - Build Environment
    - Build Tools
    - Build Deployment
    - Build Unpaired Fob, Paired Fob, and Car
- Load Devices
- Host Tools
    - Package Feature
    - Pair Fob
    - Enable Feature
    - Unlock and Start Car

Docker Architecture

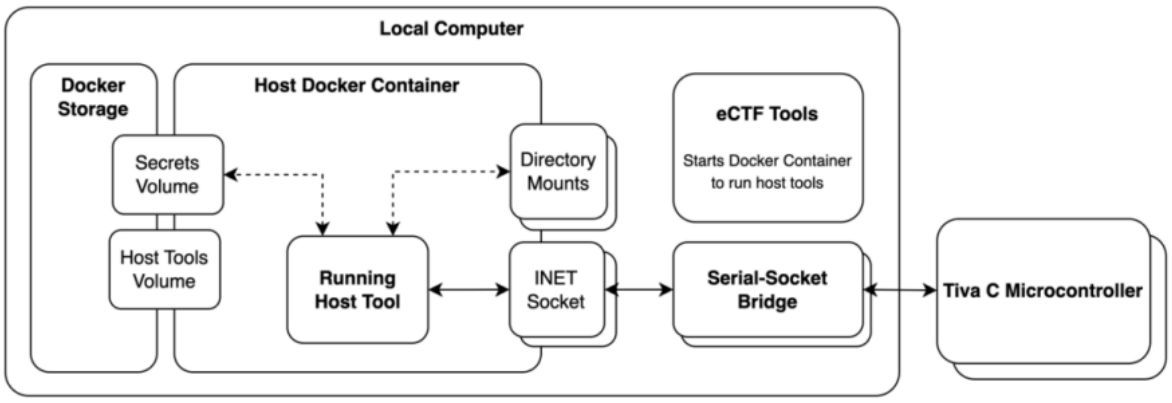

Figure 1. System Architecture

Security Requirements

1. A car should only unlock and start when the user has an authentic fob that is paired with the car
2. Revoking an attacker's physical access to a fob should also revoke their ability to unlock the associated car.
3. Observing the communications between a fob and a car while unlocking should not allow an attacker to unlock the car in the future.

4. Having an unpaired fob should not allow an attacker to unlock a car without a corresponding paired fob and pairing PIN. Without both the paired fob and associated PIN, an attacker should not be able to pair a new fob with the car. Otherwise, an attacker with just the fob or just the PIN could make their own fob.
5. A car owner should not be able to add new features to a fob that did not get packaged by the manufacturer These features must not be able to be tampered with to prevent an owner from adding a feature they have not paid for and to prevent an attacker from escalating privileges on a feature-limited fob.
6. Access to a feature packaged for one car should not allow an attacker to enable the same feature on another car For example, if your neighbor paid for an upgraded feature and the manufacturer sends them the upgrade, you should not be able to add that feature to your car without requesting it from the manufacturer.

Environment Setup

Our environment setup includes a new feature in the deployment step that generates public and private keys used for feature packaging. Private keys remain in the secret deployment directory. These are also used during package creation. Public keys are burned on the fobs, and are used to verify packages.

The rest is the same as initial setup indicated below:

Before using host tools in the following sections, these following commands will be run to set up the environment where these tools will be run. These are done in a secure facility, thus all processes run and accessed by these tools are assumed to be in an environment inaccessible to any outside entities (significant for build.deployment step).

```
python3 -m ectf_tools build.env
--design # Path to the root of the design repository
--name # Tag name given to the ectf Docker image
```

Base design builds a Docker container from the Dockerfile stored in the docker_env folder at the top level of our design. This step installs all packages and makes any environmental configurations necessary to be able to run all future build steps and host tools. After this tool is run, a Docker image is created with the name “ectf” and the tag “SYSTEM_NAME”. The system name is one of the inputs to the build environment tool. The built image will be used to run all future steps inside a Docker container.

```
python3 -m ectf_tools build.tools
--design # Path to the root of the design repository
--name # Tag name of the ectf Docker image
```

Base design compiles host tools written in a compiled language and creates the host tools volume, created during the tools build, and gets populated with the executable host tools that are copied or compiled in this step by writing to the mount location /tools_out. Future steps that use the host tools can access the built versions from this step by mounting the volume to the running container

```
python3 -m ectf_tools build.depl
--design # Path to the root of the design repository
--name # Tag name of the ectf Docker image
--deployment # Name of the deployment
```

Base design creates the secrets volume where deployment wide secrets are stored.
The functionality added at this step includes:

- ➢ Running a python script which generates a private key using ecdsa (elliptic curve digital signature algorithm) imported from python and written into deployment secrets file.
- ➢ This private key is used to authenticate new fobs being paired to existing paired fobs, and to authenticate and enable new features on a paired fob.

The following command is used to build the car and paired fob together.

```
python3 -m ectf_tools build.car_fob_pair
--design # Path to the root of the design repository
--name # Tag name of the ectf Docker image
--deployment # Name of the deployment
--car-out # Output directory for car binary
--fob-out # Output directory for fob binary
--car-name # Name of the car
--fob-name # Name of the fob
--car-id # 32b unsigned ID number for the car
--pair-pin # 6-digit pin for car
--car-unlock-secret # Unlock message in EEPROM (64B max)
--car-feature1-secret # Feature 1 message in EEPROM (64B max)
--car-feature2-secret # Feature 2 message in EEPROM (64B max)
--car-feature3-secret # Feature 3 message in EEPROM (64B max)
```

The base design creates a car and fob binary that can be flashed to a board where that board now becomes a functioning car or fob.

The functionality added at this step includes:

- ➢ Compilation of cryptography libraries
    - AES (https://github.com/kokke/tiny-AES-c)
        - to secure the unlock host tool by encrypting and decrypting in the challenge handshake protocol.
    - SHA
        - in C (https://github.com/BrianGladman/sha)
            - to create hashes passed into ECC asymmetric key exchange functions
            - to verify the pairing PIN entered during pairing process matches the hash of the manufacturer's pairing PIN
            - to verify the authenticity of the fob enabling a feature for a car.
        - in Python
            - to keep pairing pin hashed when stored in RAM rather than storing in plaintext
            - We ignored hashing the car password because if the hash itself gets leaked, it is just as good as the password.
    - ECC (https://github.com/kmackay/micro-ecc)
        - to create asymmetric key exchange when enabling a feature on a fob.
        - to authenticate the pairing process by only verifying unpaired fobs that come from the manufacturer and were not built in an outside environment.

The unpaired fob is built separately…

```
python3 -m ectf_tools build.fob
--design # Path to the root of the design repository
--name # Tag name of the ectf Docker image
--deployment # Name of the deployment (should be same in all
commands)
--fob-out # Output directory for the fob --fob-name # Name of the fob
```

Then, load a device to a corresponding board and ensure the device is freed (cyan blinking indication).. if not (and the light is solid cyan or solid green) then hold SW2 down and press reset.

```
python3 -m ectf_tools device.load_hw
--dev-in # Path to the directory containing the device
--dev-name # Name of the device
--dev-serial # Serial port to talk to the board
```

Unlock and Start Car

Security Features Addressed

The unlock feature of the car is the process used by the fob to initiate, unlock, and start the car. With a valid paired fob, the user should be able to unlock the car itself. The main() function of the car repeatedly calls the unlockCar() function. If no data is received, then the car remains locked, however, if a valid fob sends a valid unlock sequence to the car, then the car will unlock.

The first three security requirements are addressed with the unlocking and starting of the car. Those requirements are:

1. A car should only unlock and start when the user has an authentic fob that is paired with the car.
2. Revoking an attacker's physical access to a fob should also revoke their ability to unlock the associated car.
3. Observing the communications between a fob and a car while unlocking should not allow an attacker to unlock the car in the future.

The unlocking and starting a car process uses a Challenge Handshake Protocol in which a paired fob sends an encrypted nonce1 to the car using AES. Once the car receives the encrypted nonce1, it will send a message back to the fob with the encrypted nonce1 concatenated with another nonce2. The fob will then send a message with the encrypted nonce2 to the car. The fob and the car share a secret password that is used to encrypt and decrypt the messages sent back and forth between fob and car. If the nonce2 from the fob is correct the car will unlock and the car will then start.

The solutions to address each of these requirements are generally coinciding. Securing one element will in turn provide better security for the others.
Security Requirement 1 : A car should only unlock and start when the user has an authentic fob that is paired with the car
Design : The car will only unlock if presented with a valid, paired fob that contains the car-id and password. An authentic fob will have the 64 byte CAR_PASSWORD which will be securing all communications with the car during unlock. A random fob (unauthentic) will not have this password.

Security Requirement 2 Revoking an attacker's physical access to a fob should also revoke their ability to unlock the associated car.
Design : The handshake protocol in unlock() uses two randomly generated NONCE (one from car and one from fob). Hence, no transactions would be similar. Additionally, a timeout is utilized to ensure brute force is limited.

Security Requirement 3 Observing the communications between a fob and a car while unlocking should not allow an attacker to unlock the car in the future.
Design : The design for SR2 also handles SR3.

Commands Run

```
python3 -m ectf_tools run.unlock
      --name # Tag name of the ectf Docker image

--car-bridge # Bridge ID to the car board
```

Pair Fob

Security Features Addressed

Pairing a fob to a car requires a paired fob, an unpaired fob, and a pin. Our design is based on the insecure design as it already protects against two of the three possibilities we will encounter (1 having an unpaired fob only and 2 having an unpaired fob and pin only). It is already protected because, if all the unlocking information is kept in the paired fob and the paired fob is absent, then there is no chance the unpaired fob can gain the information off the paired fob. Thus, in our system a paired fob contains the id of the car it is paired to, an encrypted password used to unlock the car, and a SHA256 hash of the pin. When the host wants to pair a fob, the hash of the received pairing PIN is calculated and compared to the existent hash of the legitimate pairing PIN.

On the unpaired Fob, we store the state in the EEPROM, and lock the EEPROM with a password generated during Fob build time.

Security Requirement 4 : Having an unpaired fob should not allow an attacker to unlock a car without a corresponding paired fob and pairing PIN.
Design : The paired fob sends a randomly generated nonce to the host using a timer running in the BG. The host then appends the nonce to the pin and encrypts it with AES encryption using a shared key. The paired fob receives an encrypted message, decrypts it, checks, ensures the nonces match, and hashes the pin and compares it to the hash it has stored. Only if the hashes match, all paired fob information is sent to the unpaired fob.

Commands Run

```
python3 -m ectf_tools run.pair
      --name # Tag name of the ectf Docker image
      --unpaired-fob-bridge # Bridge ID to the unpaired fob
      --paired-fob-bridge # Bridge ID to the paired fob
      --pair-pin # Pairing PIN for paired fob
```

Package Feature

The two security requirements to address are, first, one car's feature key can not be used to activate the same feature on another car, and second, a car's feature key can not be activated without going through the manufacturer.

We address the first requirement through use of Elliptic Curve Cryptography, an advanced asymmetric algorithm where we sign a hash. The keys used for signing will be stored securely, with the public key stored on the Fob and the private key stored in the secret.h file. We will also use SHA-256 to create a hash. In the manufacture car_id and feature_number will be hashed before the entire hash is signed. This fulfills the security requirement that another car's feature key can not be used to activate the same feature on another car. We address the second requirement when the manufacturer sends the signed hash to the Host along with the feature_number (unencrypted). The Host can then send the signed hash and the feature_number to the Fob. The Fob will first verify the signed hash with the public key. This fulfills the security requirement that a car's feature key can not be activated without going through the manufacturer. Then if verified, the Fob will generate the hash of the car_id and the feature_number.

As per the example design, “packaged feature” contains :

1) Car ID
2) Feature Number

We create a **Digital Signature (DS)** for these fields and add that to the packaged feature.

To create the digital signature, we first hash the Car ID and Feature Number using SHA256 hash and then encrypt this hash using an asymmetric algorithm – ECC (Elliptic Curve Cryptography). This encrypted hash serves as a signature, which asserts that the manufacturer has created the packaged feature. The Private Key (which we say belongs to the manufacturer) for this is stored in the Host Secrets file, which makes it hidden from the attackers.

With our design, a “packaged feature” contains :

1) Car ID
2) Feature Number
3) DS(Car ID, Feature Number)

This packaged feature is given to the car/fob owner.

The fact that a user does not have access to the manufacturer’s private key makes sure that a random user is not able to create a feature signature. We have logic in the fob’s firmware (explained in the next section) to validate this signature, and reject a feature where the signature seems forged.

Commands Run

```
python3 -m ectf_tools run.package
--name # Tag name of the ectf Docker image
--deployment # Name of the deployment
--package-out # Path to output directory
--package-name # Name of the packaged feature binary file
--car-id # 32b unsigned ID number for the car
--feature-number # 32b unsigned feature number
```

Enable Feature

Security Features Addressed

As discussed in the previous section, the feature packaged by the manufacturer consists of a digital signature.

When this package is sent to the fob, the fob receives a packet containing :

1) Car ID
2) Feature Number
3) DS(Car ID, Feature Number)

If the Car ID does not match the Car ID stored on the Fob, the package is rejected.

The fob has the Manufacturer’s public key burnt in. The fob uses this public key to decrypt the Digital Signature and get a Hash (H1).
The fob then creates a SHA256 hash of the Car ID and Feature Number (H2).
The fob then compares H1 and H2, and if they match, it means that the feature package was created by the legitimate manufacturer. If the hashes don’t match, the fob rejects the new package.

Security Requirement 5 : A car owner should not be able to add new features to a fob that did not get packaged by the manufacturer.
Design : Since the “feature number” is digitally signed, an illegitimate user would not be able to create a valid signature, due to lack of the manufacturer’s private key. A package without a valid signature would be rejected by the fob.

Security Requirement 6 Access to a feature packaged for one car should not allow an attacker to enable the same feature on another car.
Design : Since the “Car ID” is digitally signed, an illegitimate user would not be able to create a package with a valid signature, with a spoofed Car ID. The fob matches the Car ID and validates the signature. This package will be rejected by the Fob.

Commands Run

```
python3 -m ectf_tools run.enable
--name # Tag name of the ectf Docker image
--fob-bridge # Bridge ID to the fob board
--package-in # Path to the input directory
--package-name # Name of the package binary file
```