Technical Specifications

PARED: Protected Automotive Remote Entry

MITRE | SOLVING PROBLEMS FOR A SAFER WORLD™

© 2023 The MITRE Corporation. All rights reserved. Approved for public release. Distribution unlimited 22-03216-5.

Table of Contents

1 SYSTEM IMPLEMENTATION ... 3
1.1 Docker Architecture ... 3
1.1.1 Docker Container ... 3
1.1.2 Docker Volumes and Bind Mounts ... 3
1.1.3 Sockets ... 3
1.2 Repository Structure ... 3
1.3 Bootloader ... 4
1.4 Requirements and Restrictions ... 4
1.4.1 Time Requirements ... 4
1.4.2 Size Requirements ... 5
1.4.3 Memory Layout ... 5
1.4.4 Flash Memory Protections ... 6
1.4.5 EEPROM Block Hiding ... 6
1.4.6 Interrupt Vector Table ... 6
2 HANDOFF SUBMISSION ... 7
3 FUNCTIONAL REQUIREMENTS ... 8
3.1 Installing the Tools Repository ... 8
3.2 Build ... 8
3.2.1 Build Environment ... 8
3.2.2 Build Tools ... 9
3.2.3 Build Deployment ... 9
3.2.4 Build Car and Paired Fob ... 10
3.2.5 Build Unpaired Fob ... 11
3.3 Load Device ... 11
3.4 Start Bridge ... 12
3.5 Host Tools ... 12
3.5.1 Pair Fob ... 12
3.5.2 Package Feature ... 13
3.5.3 Enable Feature ... 13
3.5.4 Unlock Car ... 13

© 2023 The MITRE Corporation. All rights reserved.
Approved for public release. Distribution unlimited 22-03216-5.

1 System Implementation

1.1 Docker Architecture

The reference PARED design provided by the organizers uses Docker to build container images for the host computer. The Docker container architecture has been designed to support machines with various operating systems that will interact with the physical hardware. Figure 1 shows an overview of the architecture for a generic build step or host tool invocation.

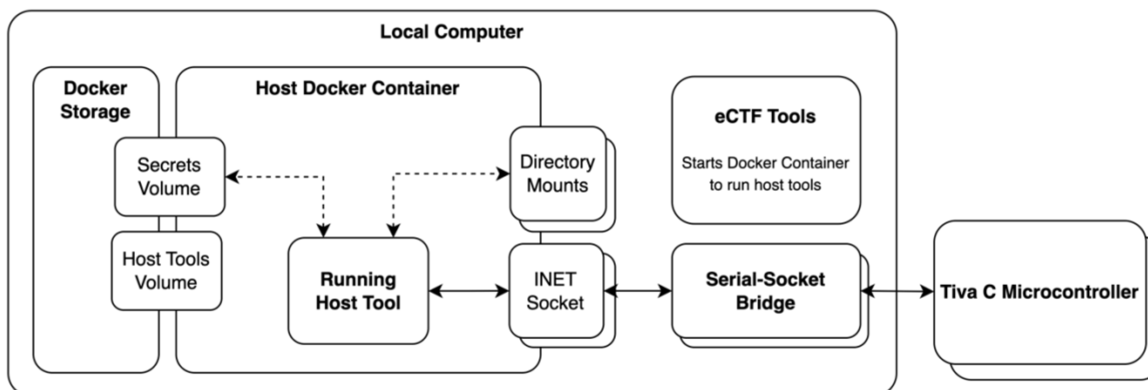

Figure 1. System Architecture

1.1.1 Docker Container

PARED build tools and host tools run in a Docker container to support multiple operating systems and to eliminate the need for other teams to manually install dependencies for other teams' designs on their local machine. The Docker image is built first, which can be thought of as a snapshot of a virtual machine with required software pre-installed.

1.1.2 Docker Volumes and Bind Mounts

Docker volumes and directory mounts are used so that the host container can maintain any state and share files with the host computer. Docker volumes create a virtual file system that can be used by multiple containers to share files. Directory mounts share a part of the host file system with the container environment, and changes made in the container persist on the host. Your PARED system has two volumes. The host tools volume, as the name suggests, stores your built host tools. The secrets volume stores any system wide secrets generated during the build deployment step. The directory mounts are used to share source code, build artifacts, and host tool outputs between the container and the local computer.

1.1.3 Sockets

Internet sockets are used in the system to bridge the UART connections from the Tiva C Microcontrollers to the host tools running inside the Docker container. Although Docker supports adding devices to the container, the socket implementation enables cross-platform support.

1.2 Repository Structure

The eCTF source code is split into two repositories:

© 2023 The MITRE Corporation. All rights reserved.
Approved for public release. Distribution unlimited 22-03216-5.

1. eCTF Tools: https://github.com/mitre-cyber-academy/2023-ectf-tools
2. Example Design: https://github.com/mitre-cyber-academy/2023-ectf-insecure-example

The pip-installable 2023-ectf-tools repository contains code used to run each team's design. **Nothing in this repository should be modified.** The organizers will use a clean copy of this repository to test your design. Therefore, any changes made by a team will be discarded. Although your submission may not modify these files, it may be useful to make changes during your own testing for debugging purposes.

The 2023-ectf-insecure-example repository contains a PARED example design that satisfies the basic functional requirements of the challenge. This should be used by teams as a reference, but **it should not be trusted to satisfy any security requirements**. Teams can use any of this code in their own designs at their own risk.

Each team must model their design after the structure in the top level of the example design repository to ensure that the tools repo can build and run their design. The following folders and files must not be moved, deleted, or renamed. However, the Dockerfile and Makefiles may be edited to implement your design.

- ***docker_env/*** – contains the Dockerfile for creating the runtime environment
  - ***build_image.Dockerfile***
- ***host_tools/*** – code for the PARED host tools
  - ***Makefile***
- ***deployment/*** – code for generating the host secrets file
  - ***Makefile***
- ***car/*** – code for the PARED car devices
  - ***Makefile***
- ***fob/*** – code for the PARED fob devices
  - ***Makefile***

1.3 Bootloader

The organizers will provide teams with an unkeyed bootloader application that will be used in the Design Phase. This will be functionally equivalent to the keyed bootloader teams will use in the Attack Phase to load other teams' designs. You are responsible for ensuring that your design does not access any regions it doesn't have access to as specified in Section 1.4.3. A design that does not follow these rules will not work with the Attack Phase bootloader. However, we don't check for invalid memory access with the Design Phase bootloader to allow teams to use a debugger.

1.4 Requirements and Restrictions

1.4.1 Time Requirements

The PARED system must complete each operation within the following time limits:

OPERATION	MAXIMUM TIME FOR COMPLETION
Boot	1 second
Pair Fob	1 second
Package Feature	1 second

© 2023 The MITRE Corporation. All rights reserved.

Approved for public release. Distribution unlimited 22-03216-5.

Enable Feature	1 second
Unlock Car	1 second

1.4.2 Size Requirements

The PARED system must comply with the following size restrictions:

COMPONENT	SIZE
Car Firmware	Max 110 KB
Car EEPROM Data	Max 1792 bytes
Fob Firmware	Max 110 KB
Fob EEPROM Data	Max 1792 bytes
Programming PIN	6 digits
Feature Number	32-bit unsigned integer
Feature Message	Max 64 bytes
Unlock Message	Max 64 bytes
Car ID	32-bit unsigned integer

1.4.3 Memory Layout

Your PARED system must compile any firmware (car and fob) such that its first instruction is at address 0x00008000. Your system must not use any Flash memory located below this address[1]. Additionally, the last 256 bytes of EEPROM are reserved for messages that get printed over UART when the car is unlocked and when specific features are enabled.

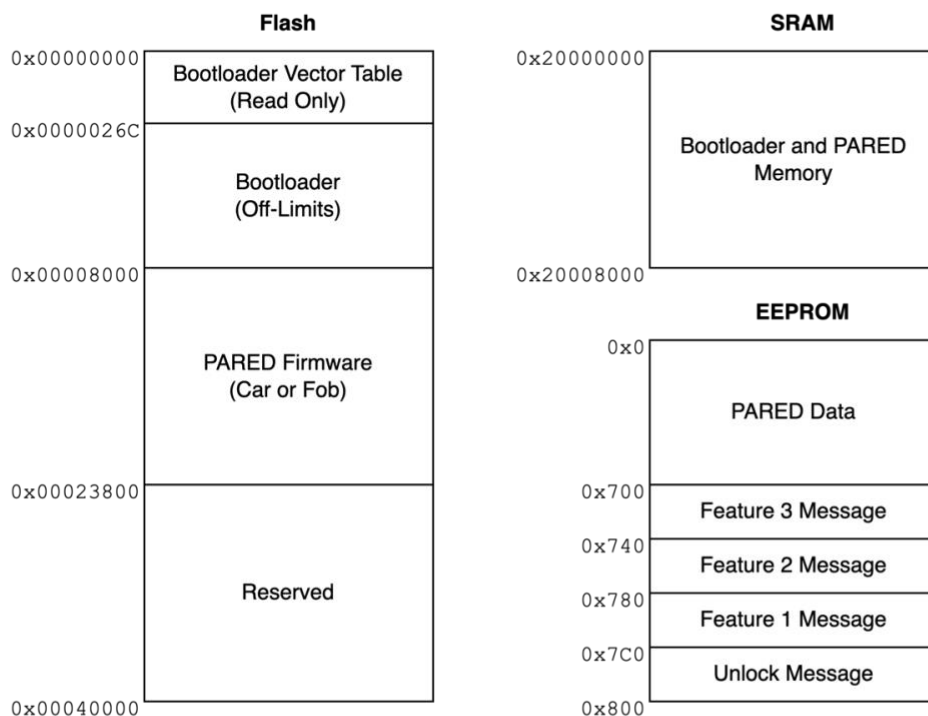

Figure 2. PARED Memory Layout

[1] See 1.4.6 for how this impacts interrupts

© 2023 The MITRE Corporation. All rights reserved.
Approved for public release. Distribution unlimited 22-03216-5.

1.4.4 Flash Memory Protections

PARED systems ***may not*** permanently commit Flash memory write protections. This is to ensure that keyed Attack Phase devices can be used for multiple designs. PARED systems ***may*** set Flash memory write protections on each boot as long as it does not prevent the design from working after any reset.

1.4.5 EEPROM Block Hiding

PARED systems ***may*** use EEPROM block hiding as long as it does not prevent the design from working after any reset.

1.4.6 Interrupt Vector Table

If a team wants to use interrupts in their PARED system, they must place a copy of the vector table in SRAM or Flash after the start of their system. This functionality is implemented in the Tivaware driver library[2].

[2] `tivaware/driverlib/interrupt.c:IntRegister()`

© 2023 The MITRE Corporation. All rights reserved.
Approved for public release. Distribution unlimited 22-03216-5.

2 Handoff Submission

When submitting your design to the organizers for verification, your design must reside in a public facing git repository (github, gitlab, etc.). When you are ready to submit, the organizers will provide an account that you must give access to your source code.

Each version of your design that is submitted for testing must be tagged with a version number starting from “v1.0”. If the tests fail when the organizers are validating your submission, you will be informed and given a log of the test process. When submitting the next version of your code, if your system failed functional testing and requires a resubmit, you must increase the major version number (e.g., “v1.0” to “v2.0”) before resubmitting. If the organizers find a small bug/discrepancy that doesn’t trigger a full resubmit, you must increase the minor version number (e.g., “v1.0” to “v1.1”) before testing resumes.

This year, we are introducing an automated hardware testing server for teams to use as they develop their designs. To submit a design, you will post the git URL of your repository in the #testing channel on Slack (using a Slack bot), and our server will load your design onto physical hardware, run a series of functional tests, and post the results to your team channel. This will run the same set of tests we will use to determine if your design is functional. Once this service is available to teams, we will announce more details in Slack. To ensure that teams know how to use this testing service, one of the Design Phase Flags is to submit the example design to the testing server[3].

When testing a submission, the evaluation process will automatically replace any folders that are not allowed to be modified with a clean version. Therefore, you may change files in non-modifiable folders for the purposes of testing your design, but any changes or new files will not be included when the organizers test your submission. See the repository structure for a list of folders that are not modifiable.

When you are ready to submit your final design, you should notify the organizers in your team channel, and include the testing results from the automated testing server. In addition to verifying the results of the tests, the organizers will review your source code to ensure that all rules have been followed such as not incorporating permanent lockouts in your design.

[3] We have set a preliminary deadline for teams to submit this flag. If we are not able to release the testing service within a reasonable amount of time for teams to complete this task, we will extend the deadline.

© 2023 The MITRE Corporation. All rights reserved.

Approved for public release. Distribution unlimited 22-03216-5.

3 Functional Requirements

The Rules Document introduced each functional step in PARED, and how they should interact with the microcontrollers. This section formally defines the inputs and outputs of each step. Each step in the system is run by invoking a python script in the tools repository. The python script will handle the execution of Docker build commands, Makefile calls, and Docker run commands.

The following sections will describe the API for each tool. In these sections, the *Top Level Call* is the python script you will run from a terminal with your python virtual environment activated. There are also sections for a *Makefile Call* or a *Tool Call*. These calls are handled by the python script at the top level and are executed in the Docker container. Finally, you will see sections titled *Environment of Tool Within Docker*. These sections will show what volumes and directory mounts are attached to the running container, and the folder location they are mounted to inside the container. Additionally, it shows the working directory for the calls that get made when the container is run. This information is important when you are writing your tools to access or store data in these shared folders.

Do not copy and paste commands from this document!
Invisible or invalid Unicode characters may be copied over and give errors.

3.1 Installing the Tools Repository

To run all the steps in your design, you will need to install the tools repository provided by the organizers. Before installing the repository, create a python virtual environment, activate it, and update the pip package to the latest version. Then, clone the tools repository to a location you choose. With your python virtual environment activated, install the repository by running the following command:

```
python3 -m pip install -e <path>/2023-ectf-tools
```

This command will install the tools repository as an editable python module (the -e flag). With this module installed, it will create an executable python module called `ectf_tools` that is used to invoke each of the steps to build and run your design.

3.2 Build

The build steps are used to create your Docker image, host tools, host secrets, and device images.

Docker must be running to execute these steps.

3.2.1 Build Environment

The build environment step builds a Docker container from the Dockerfile stored in the docker_env folder at the top level of your design. This step should install all packages and make 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.

© 2023 The MITRE Corporation. All rights reserved.
Approved for public release. Distribution unlimited 22-03216-5.

Top Level Call

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

3.2.2 Build Tools

The build tools step is an opportunity to compile host tools written in a compiled language. Tools written in interpreted languages, like Python used in the reference design, simply need to be copied over to the output directory. When this tool executes, a new docker volume is created with the name “ectf.SYSTEM_NAME.tools.vol”. The SYSTEM_NAME must be the same one specified in the previous environment build. This volume 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. To change the behavior of this step, edit the Makefile in the /host_tools folder at the top level of your design.

Top Level Call

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

Makefile Call

```
make TOOLS_OUT_DIR=/tools_out
```

All host tools should be ready to be executed from the root of /tools_out.

Environment of Tool Within Docker

Working directory: /tools_in

Source directory or volume	Location in build container
<DESIGN>/host_tools (readonly)	/tools_in (readonly)
ectf.<SYSTEM_NAME>.tools.vol	/tools_out

3.2.3 Build Deployment

All deployment-wide secrets (i.e., ones that should be shared across cars and/or fobs) should be built in the build deployment step. The nature and structure of these secrets is completely up to your team. When this tool executes, a new docker volume is created with the name “ectf.<SYSTEM_NAME>.<DEPL>.secrets.vol”. Any secrets needed by your deployment should be stored in this volume, which is mounted to the container at /secrets. To change the behavior of this step, edit the Makefile in the /deployment folder at the top level of your design.

Top Level Call

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

Makefile Call

```
make SECRETS_DIR=/secrets
```

© 2023 The MITRE Corporation. All rights reserved.
Approved for public release. Distribution unlimited 22-03216-5.

Environment of Tool Within Docker

Working directory: /depl_in

Source directory or volume	Location in build container
<DESIGN>/deployment (readonly)	/depl_in (readonly)
ectf.<SYSTEM_NAME>.<DEPL>.secrets.vol	/secrets

3.2.4 Build Car and Paired Fob

This step should build a new car with a single fob paired with it. The PIN to pair an additional unpaired fob to work with this car and the secrets that are written to the last 256B of EEPROM for use during the unlock step are defined here.

Top Level Call

```
python3 -m ectf_tools build.car_fob_pair
       --design <DESIGN>                 # Path to the root of the design repository
       --name <SYSTEM_NAME>              # Tag name of the ectf Docker image
       --deployment <DEPL>               # Name of the deployment
       --car-out <CAR_OUT>               # Output directory for car binary
       --fob-out <FOB_OUT>               # Output directory for fob binary
       --car-name <CAR_NAME>             # Name of the car
       --fob-name <FOB_NAME>             # Name of the fob
       --car-id <ID>                     # 32b unsigned ID number for the car
       --pair-pin <PIN>                  # 6-digit pin for car
       --car-unlock-secret <UL_SECRET>   # Unlock message in EEPROM (64B max)
       --car-feature1-secret <F1_SECRET> # Feature 1 message in EEPROM (64B max)
       --car-feature2-secret <F2_SECRET> # Feature 2 message in EEPROM (64B max)
       --car-feature3-secret <F3_SECRET> # Feature 3 message in EEPROM (64B max)
```

Car Build Step

Makefile Call

```
make CAR_ID=<ID> BIN_PATH=/dev_out/<CAR_NAME>.bin SECRETS_DIR=/secrets
       ELF_PATH=/dev_out/<CAR_NAME>.elf EEPROM_PATH=/dev_out/<CAR_NAME>.eeprom
```

The resulting car binary should be written in binary form and elf form to /dev_out/<CAR_NAME>.bin and /dev_out/<CAR_NAME>.elf, respectively. Additionally, its EEPROM data should be written in binary form to /dev_out/<CAR_NAME>.eeprom.

Environment of Tool Within Docker

Working directory: /dev_in

Source directory or volume	Location in build container
<DESIGN>/car (readonly)	/dev_in (readonly)
<CAR_OUT>	/dev_out
ectf.<SYSTEM_NAME>.<DEPL>.secrets.vol	/secrets

Fob Build Step

Makefile Call

```
make CAR_ID=<ID> PROGRAM_PIN=<PIN> SECRETS_DIR=/secrets
       BIN_PATH=/dev_out/<FOB_NAME>.bin ELF_PATH=/dev_out/<FOB_NAME>.elf
       EEPROM_PATH=/dev_out/<FOB_NAME>.eeprom
```

© 2023 The MITRE Corporation. All rights reserved.
Approved for public release. Distribution unlimited 22-03216-5.

The resulting fob binary should be written in binary form and elf form to /dev_out/<FOB_NAME>.bin and /dev_out/<FOB_NAME>.elf. The EEPROM data should be written in binary form to /dev_out/<FOB_NAME>.eeprom.

Environment of Tool Within Docker

Working directory: /dev_in

Source directory or volume	Location in build container
<DESIGN>/fob (readonly)	/dev_in (readonly)
<FOB_OUT>	/dev_out
ectf.<SYSTEM_NAME>.<DEPL>.secrets.vol	/secrets

3.2.5 Build Unpaired Fob

This step should build a single unpaired fob. The fob will not be connected to any car and should be able to be paired to any car with the respective paired fob and pairing PIN.

Top Level Call

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

Makefile Call

```
make SECRETS_DIR=/secrets BIN_PATH=/dev_out/<FOB_NAME>.bin
       ELF_PATH=/dev_out/<FOB_NAME>.elf EEPROM_PATH=/dev_out/<FOB_NAME>.eeprom
```

The resulting fob binary should be written in binary form and elf form to /dev_out/<FOB_NAME>.bin and /dev_out/<FOB_NAME>.elf. The EEPROM data should be written in binary form to /dev_out/<FOB_NAME>.eeprom.

Environment of Tool Within Docker

Working directory: /dev_in

Source directory or volume	Location in build container
<DESIGN>/fob (readonly)	/dev_in (readonly)
<FOB_OUT>	/dev_out
ectf.<SYSTEM_NAME>.<DEPL>.secrets.vol	/secrets

3.3 Load Device

This step loads a car or fob onto the board. This tool is provided to you by the organizers and may not be modified. Before running this command, you will need to get the hardware ready for a firmware update by holding down SW2 while resetting the board. You will know the microcontroller is ready for an update if there is a blinking cyan LED on the board.

Top Level Call

```
python3 -m ectf_tools device.load_hw
       --dev-in <DEV_DIR>    # Path to the directory containing the device
       --dev-name <DEV_NAME> # Name of the device
```

© 2023 The MITRE Corporation. All rights reserved.
Approved for public release. Distribution unlimited 22-03216-5.

```
       --dev-serial <SERIAL> # Serial port to talk to the board
```

After loading, the LED on the board should be solid cyan, signifying that the image was installed and ready to boot. After a power cycle, the board should boot the firmware and the LED should turn solid green.

3.4 Start Bridge

This step starts the serial bridge that allows host tools running in Docker containers to talk to the board. The Bridge ID defines a network port to talk to the host tools 1337 higher than the ID.

This tool should either be run in the background or in a separate terminal.

The bridge must continue running for your host tools to communicate with the microcontrollers from the Docker container, and you need to start the bridge for each board used by the host tools.

Top Level Call

```
python3 -m ectf_tools device.bridge
       --bridge-id <BRIDGE>  # Bridge ID
       --dev-serial <SERIAL> # Serial port to talk to the board
```

3.5 Host Tools

The host tools are used to interact with your PARED design. The following sections detail the arguments that must be provided to the top level ectf_tools module to run your host tools in a Docker container based on the Docker Image created in the Build Environment step.

Docker must be running to execute these steps.

3.5.1 Pair Fob

This step initiates a pairing session between an unpaired fob and a paired fob. This step should only work if the correct pairing PIN is provided. After running, the newly paired fob should be able to pair other unpaired fobs using the same PIN.

Top Level Call

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

Tool Call

```
./program_tool --unpaired-fob-bridge <UN_BRIDGE + 1337>
       --paired-fob-bridge <PAIR_BRIDGE + 1337> --pair-pin <PIN>
```

Environment of Tool Within Docker

Working directory: /tools_out

Source directory or volume	Location in build container
ectf.<SYSTEM_NAME>.tools.vol (readonly)	/tools_out (readonly)

© 2023 The MITRE Corporation. All rights reserved.

Approved for public release. Distribution unlimited 22-03216-5.

3.5.2 Package Feature

This step generates a packaged feature as a binary file. It utilizes the feature number and the car ID number to build the file. The packaged feature can be used later in the enable feature step.

Top Level Call

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

Tool Call

```
./package_tool –package-name <PACKAGE_NAME> --car-id <CAR_ID>
       --feature-number <FEATURE_NUMBER>
```

Environment of Tool Within Docker

Working directory: /tools_out

Source directory or volume	Location in build container
<PACKAGE_OUT>	/package_dir
ectf.<SYSTEM_NAME>.tools.vol (readonly)	/tools_out (readonly)
ectf.<SYSTEM_NAME>.<DEPL>.secrets.vol	/secrets

3.5.3 Enable Feature

This step enables a previously packaged feature by loading it onto a paired fob that works with the car the feature was packaged for.

Top Level Call

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

Tool Call

```
./enable_tool --fob-bridge <FOB_BRIDGE + 1337> --package-name <PACKAGE_NAME>
```

Environment

Working directory: /tools_out

Source directory or volume	Location in build container
<PACKAGE_IN>	/package_dir
ectf.<SYSTEM_NAME>.tools.vol (readonly)	/tools_out (readonly)

3.5.4 Unlock Car

This step unlocks a car using a paired fob. If the unlock succeeds, the car must print out a message stored in the last 64 bytes of EEPROM over UART. Additionally, the car must print out

© 2023 The MITRE Corporation. All rights reserved.

Approved for public release. Distribution unlimited 22-03216-5.

the corresponding message for each enabled feature stored in EEPROM over UART. The locations of these messages are shown in Section 1.4.3.

Top Level Call

```
python3 -m ectf_tools run.unlock
       --name <SYSTEM_NAME>      # Tag name of the ectf Docker image
       --car-bridge <CAR_BRIDGE> # Bridge ID to the car board
```

Tool Call

```
./unlock_tool --car-bridge <CAR_BRIDGE + 1337>
```

Environment of Tool Within Docker

Working directory: /tools_out

Source directory or volume	Location in build container
ectf.<SYSTEM_NAME>.tools.vol (readonly)	/tools_out (readonly)

© 2023 The MITRE Corporation. All rights reserved.
Approved for public release. Distribution unlimited 22-03216-5.