MITRE eCTF 2024

Final Design Document

TNTech eCTF 2024

This repository holds the modified insecure reference design for the 2024 eCTF competition. The design is based on the [Analog Devices MAX78000](#) microcontroller. The design is based on the [eCTF 2024 Reference Design](#).

Building

Prerequisites

- [Nix](#)

```
git clone https://github.com/tntechcsc/MITRE_eCTF_2024
cd MITRE_eCTF_2024
nix-shell
poetry install
```

Environment Variables

You need to create a `.env` file in the root of the project with the following variables:

```
PIN="123456"
TOKEN="abcdefgh12345678"
COMPONENT_CNT="2"
COMPONENT_IDS="0x11111124, 0x11111125"
BOOT_MESSAGE="Hello world"
COMPONENT_BOOT_MESSAGE="Component booted up"
ATTESTATION_LOCATION="USA"
ATTESTATION_DATE="08/08/08"
ATTESTATION_CUSTOMER="eCTF"
```

Custom Build Pipeline

The `build.py` Python script is used to facilitate the build process. It is a wrapper around the eCTF tools and provides a more user-friendly interface. The script is used to build, flash, and run the host tools.

```
python3 build.py
  - usage: build.py [-h] [-b {ap,comp}] [-c COMPONENT] [-f {ap,comp}] [-s
SERIAL] [--pin PIN] [--list] [--attest] [--boot] [--debug]
```

Build AP

```
python3 build.py -b ap
```

Build Component

```
python3 build.py -b comp
  - Must specify component ID (--component). Valid options are: 0x11111124,
0x11111125
```

```
python3 build.py -b comp -c 0x11111124
```

Flash AP

```
python3 build.py -f ap
  - Must specify serial port of AP (--serial).
```

```
python3 build.py -f ap --serial /dev/ttyUSB0
```

Flash Component

```
python3 build.py -f comp
  - Must specify serial port of component (--serial).
  - Must specify component ID (--component). Valid options are: 0x11111124,
0x11111125
```

```
python3 build.py -f comp --serial /dev/ttyUSB0 --component 0x11111124
```

List Components

```
python3 build.py --list
  - Must specify serial port of AP (--serial).
```

```
python3 build.py --list --serial /dev/ttyUSB0
```

Attest Component

```
python3 build.py --attest
  - Must specify serial port of AP (--serial).
  - Must specify component ID (--component). Valid options are: 0x11111124,
0x11111125
  - Must specify PIN for component (--pin).
```

```
python3 build.py --attest --serial /dev/ttyUSB0 --component 0x11111124 --pin
123456
```

Boot System

```
python3 build.py --boot
  - Must specify serial port of AP (--serial).
```

```
python3 build.py --boot --serial /dev/ttyUSB0
```

Debug

```
python3 build.py --debug
```

Our Design

For security requirements, see the [Security Implementations] section.

- `build.py` - This script is used to facilitate the build process. It is a wrapper around the eCTF tools and provides a more user-friendly interface.
- `testing.py` - A Python script for quick debugging and testing. Not necessary for the build process.
- `.env` - Environment variables for the build process used by `build.py`.
- `deployment` - Used for generating secrets available to both the AP and Components
  - `generator.py` - This script generates `HASH_PEPPER`, `AP_FIRMWARE_TOKEN`, and `ENCRYPTION_KEY` and stores them in `global_secrets.h`.
- `application_processor` - Code for the application processor
  - `generator.py` - This script hashes the provisioned `AP_PIN` and `AP_TOKEN` with the pepper and stores the hashes in `ectf_params.h`.
  - `inc` - Directory with c header files
    - `security.h` - This file contains the constant time string and memory comparison functions.
    - `simple_crypto.h` - This file contains the encryption and decryption functions.
  - `src` - Directory with c source files
    - `security.c` - This file contains constant time string and memory comparison functions.
    - `simple_crypto.c` - This file contains the encryption and decryption functions.
- `component` - Code for the components
  - `inc` - Directory with c header files
    - `security.h` - This file contains the constant time string and memory comparison functions.
    - `simple_crypto.h` - This file contains the encryption and decryption functions.
  - `src` - Directory with c source files
    - `security.c` - This file contains constant time string and memory comparison functions.
    - `simple_crypto.c` - This file contains the encryption and decryption functions.

Function Definitions

`bool constant_strcmp(const char *s1, const char *s2)`

- Located in `src/security.c` and `inc/security.h`
- This function compares two strings in constant time.

- Returns `true` if the strings are equal, `false` otherwise.

`bool constant_memcmp(const void *s1, const void *s2, size_t n)`

- Located in `src/security.c` and `inc/security.h`
- This function compares two memory buffers in constant time.
- Returns `true` if the memory buffers are equal, `false` otherwise.

`int sha256_hash(void *data, size_t len, uint8_t *hash_out)`

- Located in `src/simple_crypto.c` and `inc/simple_crypto.h`
- This function hashes the input data using the SHA-256 hash function and stores the hash in `hash_out`.
- Returns `0` on success, non-zero on failure.

`int create_encrypted_packet(uint8_t *plaintext, size_t plaintext_len, uint8_t *key, uint8_t *packet)`

- Located in `src/simple_crypto.c` and `inc/simple_crypto.h`
- This function encrypts the plaintext using the ChaCha20-Poly1305 AEAD cipher and stores the encrypted packet in `packet`.
- Returns `0` on success, non-zero on failure.

`int decrypt_encrypted_packet(uint8_t *packet, uint8_t *key, uint8_t *plaintext)`

- Located in `src/simple_crypto.c` and `inc/simple_crypto.h`
- This function decrypts the encrypted packet using the ChaCha20-Poly1305 AEAD cipher and stores the plaintext in `plaintext`.
- Returns `0` on success, non-zero on failure.

`void start_timer()`

- Located in `src/application_processor.c`
- This function sets a global variable (`start_time`) to the current time using MSDK's `MXC_RTC_GetSecond()`

`bool timer_expired()`

- Located in `src/application_processor.c`

- This function is used to check if the timer has expired by comparing the current time with `start_time`. If the difference is greater than or equal to `TIMEOUT_SECONDS`, the timer has expired.
- Returns `true` if the timer has expired, `false` otherwise.

Security Implementations

Security Requirement 1

The Application Processor (AP) should only boot if all expected Components are present and valid.

During the build deployment, we provision a randomly generated `COMPONENT_FIRMWARE_TOKEN` that is stored in `global_secrets.h`. During the component boot process, a component will send back its token and boot message to the AP. The AP will extract the token from the message and compare it with the provisioned token. If the tokens match, the AP will consider the component valid. If the tokens do not match, the AP will consider the component invalid and fail to boot.

Security Requirement 2

Components should only boot after being commanded to by a valid AP that has confirmed the integrity of the device.

During the build deployment, we provision a randomly generated `AP_FIRMWARE_TOKEN` that is stored in `global_secrets.h`. This token is used to ensure that the AP is the only device that can command the components. When the AP sends pre-boot commands to the components, it includes the token. The components will extract the token from the message and compare it with the provisioned token. If the tokens match, the component will execute the command. If the tokens do not match, the component will ignore the command.

Security Requirement 3

The Attestation PIN and Replacement Token should be kept confidential.

During build deployment, we generate a random pepper which is stored in `global_secrets.h` as `HASH_PEPPER`. Then, during the build AP process, we hash the PIN and token with the pepper and store it in `application_processor/inc/ectf_params.h`. The goal of this is to ensure that the PIN and token are not stored in plaintext on the AP. The pepper helps to prevent rainbow table attacks.

Security Requirement 4

Component Attestation Data should be kept confidential. Attestation Data should only be returned by the AP for a valid Component if the user is able to provide the correct Attestation PIN.

When a user requests attestation or replacement, the AP will take the user's input and hash it with the pepper and compare it with the stored hash. If the hashes match, the AP will request the attestation or replacement from the component.

Security Requirement 5

The integrity and authenticity of messages sent and received using the post-boot MISC secure communications functionality should be ensured.

We implemented two functions in `src/simple_crypto.c` to encrypt and decrypt messages. Specifically, we used the `wolfSSL` library and the ChaCha20-Poly1305 AEAD cipher. This helps ensure not only the confidentiality of the messages but also the integrity and authenticity.

We created the following packet format:

```
+------------------------------------------+
| Total Length |     IV     |  Auth. Tag  |
+--------------+------------+-------------+
|    1 Byte    |  12 Bytes  |  16 Bytes   |
+------------------------------------------+
|              Ciphertext                 |
|                 ...                     |
+------------------------------------------+
```

We created two helper functions to create and decrypt the packets (`create_encrypted_packet` and `decrypt_packet`). The `create_encrypted_packet` function takes the plaintext, length of the plaintext, key, and a buffer to store the encrypted packet. The `decrypt_packet` function takes the ciphertext, key, and a buffer to store the decrypted plaintext. These helper functions handle random IV generation, packet parsing, and error handling.

So, when encrypting a message, the following steps are taken:

1. Determine the total length of the message. (Length of plaintext + 1 + 12 + 16)
2. Create a encrypted buffer of the total length.
3. Call `create_encrypted_packet(plaintext, len, key, encrypted_buffer)`

When decrypting a message, the following steps are taken:

1. Determine the length of the ciphertext. (total length - 1 - 12 - 16)
2. Create a plaintext buffer of the length of the ciphertext.
3. Call `decrypt_packet(ciphertext, key, plaintext_buffer)`

The `ENCRYPTION_KEY` we use is provisioned during the build deployment process and stored in `global_secrets.h`. This key is used to encrypt and decrypt messages between the AP and components.

Other Security Considerations

1. We implemented a four second delay between failed attempts to request attestation or replacement. This is to help mitigate brute force attacks.
2. We implemented constant time string and memory comparison functions (`src/security.c`) to help mitigate timing attacks on sensitive comparisons such as PIN and token comparisons.
3. We went beyond authenticating AP/Component in the boot command functionality. All commands require valid AP and Components, otherwise the commands will fail. This is to ensure, for example, a counterfeit AP is not able to perform component replacement.
4. Instead of using the provided MD5 hash function, we switched to the SHA-256 hash function for hashing the PIN and token. This is to ensure a more secure hash function is used.
5. We attempted to remove potential buffer overflows. For example, in the `recv_input()` function, it blindly called `gets()` and we switched to `fgets()` with an explicit buffer size. We also added bound checking in `simple_i2c_peripheral.c` to ensure the the buffer being written is not larger than `MAX_I2C_MESSAGE_LEN`.

Everything below this is from the original README

Layout

- `application_processor` - Code for the application processor
  - `project.mk` - This file defines project specific variables included in the Makefile
  - `Makefile` - This makefile is invoked by the eCTF tools when creating a application processor
  - `inc` - Directory with c header files
  - `src` - Directory with c source files
  - `wolfssl` - Location to place wolfssl library for included Crypto Example
- `deployment` - Code for deployment secret generation
  - `Makefile` - This makefile is invoked by the eCTF tools when creating a deployment
  - You may put other scripts here to invoke from the Makefile

- `ectf_tools` - Host tools and build tools - DO NOT MODIFY ANYTHING IN THIS DIRECTORY
  - `attestation_tool.py` - Runs attestation command on application processor
  - `boot_tool.py` - Boots the application processor and sensors
  - `list_tool.py` - Lists what sensors are currently online
  - `replace_tool.py` - Replaces a sensor id on the application processor
  - `build tools` - Tools to build
- `component` - Code for the components
  - `project.mk` - This file defines project specific variables included in the Makefile
  - `Makefile` - This makefile is invoked by the eCTF tools when creating a component
  - `inc` - Directory with c header files
  - `src` - Directory with c source files
  - `wolfssl` - Location to place wolfssl library for included Crypto Example
- `shell.nix` - Nix configuration file for Nix environment
- `custom_nix_pkgs` - Custom derived nix packages
  - `analog-openocd.nix` - Custom nix package to build Analog Devices fork of OpenOCD

Usage and Requirements

This repository contains two main elements: firmware source code and tooling.

Firmware is built through the included eCTF Tools. These tools invoke the Makefiles in specific ways in the provided Nix environment. Firmware compiling should be executed through these included tools.

Source code and tooling is provided that runs directly on the host. All of these tools are created in Python. The tools can be easily installed with the use of Poetry. Once inside of the activated Nix environment, run `poetry install` to initialize the Poetry environment. These tools can be invoked either through `poetry run {toolname}` or by activating the poetry environment
with `poetry shell` and then running as standard python programs.

Environment Build

The environment is built with a Nix, which should install all packages necessary for running the design in a reproducible fashion. The environment is automatically built when an eCTF Build Tool is run. If building `analog_openocd.nix` this step may take some time to complete.

Development can be prototyped by launching into the Nix environment through `nix-shell`.

Host Tools

Host Tools for the 2024 competition do not need to be modified by teams at any point. Your design
should work with the standardized interface between host and MISC system. The host tools will pass any required arguments to the MISC system and receive all relevant output.

Deployment

When creating a deployment, the Makefile within the `deployment` folder of the design repo will be executed. This is the only stage in which information can be shared between separate portions of the build (e.g. components and application processors). A clean target should be implemented in this Makefile to allow for elimination of all generated secrets.

Application Processor and Component

When building the application processor and components, the `Makefile` with the respective directories will be invoked. The eCTF Tools will populate parameters into a C header file `ectf_params.h` within the design directory. Examples of these header files can be found in the respective main source files for the application processor and component.

Using the eCTF Tools

Building the deployment

This will run the `Makefile` found in the deployment folder using the following inputs:

```
ectf_build_depl --help
usage: eCTF Build Deployment Tool [-h] -d DESIGN

Build a deployment using Nix

options:
  -h, --help            show this help message and exit
  -d DESIGN, --design DESIGN
                        Path to the root directory of the included design
```

Example Utilization

```
ectf_build_depl -d ../ectf-2024-example
```

Building the Application Processor

This will run the `Makefile` found in the application processor folder using the following inputs:

```
ectf_build_ap --help
usage: eCTF Build Application Processor Tool [-h] -d DESIGN -on OUTPUT_NAME
[-od OUTPUT_DIR] -p P
                                             -b BOOT_MESSAGE

Build an Application Processor using Nix

options:
  -h, --help            show this help message and exit
  -d DESIGN, --design DESIGN
                        Path to the root directory of the included design
  -on OUTPUT_NAME, --output-name OUTPUT_NAME
                        Output prefix of the built application processor
binary Example 'ap' -> a
  -od OUTPUT_DIR, --output-dir OUTPUT_DIR
                        Output name of the directory to store the result:
default: .
  -p PIN, --pin PIN     PIN for built application processor
  -t TOKEN, --token TOKEN
                        Token for built application processor
  -c COMPONENT_CNT, --component-cnt COMPONENT_CNT
                        Number of components to provision Application
Processor for
  -ids COMPONENT_IDS, --component-ids COMPONENT_IDS
                        Component IDs to provision the Application Processor
for
  -b BOOT_MESSAGE, --boot-message BOOT_MESSAGE
                        Application Processor boot message
```

Example Utilization

```
ectf_build_ap -d ../ectf-2024-example -on ap --p 123456 -c 2 -ids
"0x11111124, 0x11111125" -b "Test boot message" -t 0123456789abcdef -od
build
```

Building the Component

```
ectf_build_comp --help
usage: eCTF Build Application Processor Tool [-h] -d DESIGN -on OUTPUT_NAME
[-od OUTPUT_DIR] -id COMPONENT_ID -b BOOT_MESSAGE -al
                                             ATTESTATION_LOCATION -ad
ATTESTATION_DATE -ac ATTESTATION_CUSTOMER

Build an Application Processor using Nix

options:
  -h, --help            show this help message and exit
  -d DESIGN, --design DESIGN
                        Path to the root directory of the included design
  -on OUTPUT_NAME, --output-name OUTPUT_NAME
                        Output prefix of the built application processor
binary Example 'ap' -> ap.bin, ap.elf, ap.img
  -od OUTPUT_DIR, --output-dir OUTPUT_DIR
                        Output name of the directory to store the result:
default: .
  -id COMPONENT_ID, --component-id COMPONENT_ID
                        Component ID for the provisioned component
  -b BOOT_MESSAGE, --boot-message BOOT_MESSAGE
                        Component boot message
  -al ATTESTATION_LOCATION, --attestation-location ATTESTATION_LOCATION
                        Attestation data location field
  -ad ATTESTATION_DATE, --attestation-date ATTESTATION_DATE
                        Attestation data date field
  -ac ATTESTATION_CUSTOMER, --attestation-customer ATTESTATION_CUSTOMER
                        Attestation data customer field
```

Example Utilization

```
ectf_build_comp -d ../ectf-2024-example -on comp -od build -id 0x11111125 -b
"Component boot" -al "McLean" -ad "08/08/08" -ac "Fritz"
```

Flashing

Flashing the MAX78000 is done through the eCTF Bootloader. You will need to initially flash the eCTF Bootloader onto the provided hardware.
This can be done easily by dragging and dropping the provided bootloader (for design phase: `insecure.bin`) to the DAPLink interface. DAPLink will show up as an external drive

when connected to your system. Succesfull installation would make a blue LED flash on the board.

To flash a specific bootloader image on the board (AP or Components), use `ectf_update`.

```
ectf_update [-h] --infile INFILE --port PORT

optional arguments:
  -h, --help       show this help message and exit
  --infile INFILE  Path to the input binary
  --port PORT      Serial port
```

Example Utilization

```
ectf_update --infile example_fw/build/firmware.img --port /dev/ttyUSB0
```

Host Tools

List Tool

The list tool applies the required list components functionality from the MISC system. This is availble on the
PATH within the Poetry environment as `ectf_list`.

```
ectf_list -h
usage: eCTF List Host Tool [-h] -a APPLICATION_PROCESSOR

List the components connected to the medical device

options:
  -h, --help            show this help message and exit
  -a APPLICATION_PROCESSOR, --application-processor APPLICATION_PROCESSOR
                        Serial device of the AP
```

Example Utilization

```
ectf_list -a /dev/ttyUSB0
```

Boot Tool

The boot tool boots the full system. This is available on the PATH within the Poetry environment as `ectf_boot`

```
ectf_boot --help
usage: eCTF Boot Host Tool [-h] -a APPLICATION_PROCESSOR

Boot the medical device

options:
  -h, --help            show this help message and exit
  -a APPLICATION_PROCESSOR, --application-processor APPLICATION_PROCESSOR
                        Serial device of the AP
```

Example Utilization

```
ectf_boot -a /dev/ttyUSB0
```

Replace Tool

The replace tool replaces a provisioned component on the system with a new component. This is available on the PATH within the Poetry environment as `ectf_replace`.

```
ectf_replace --help
usage: eCTF Replace Host Tool [-h] -a APPLICATION_PROCESSOR -t TOKEN -i
COMPONENT_IN -o COMPONENT_OUT

Replace a component on the medical device

options:
  -h, --help            show this help message and exit
  -a APPLICATION_PROCESSOR, --application-processor APPLICATION_PROCESSOR
                        Serial device of the AP
  -t TOKEN, --token TOKEN
                        Replacement token for the AP
  -i COMPONENT_IN, --component-in COMPONENT_IN
                        Component ID of the new component
  -o COMPONENT_OUT, --component-out COMPONENT_OUT
                        Component ID of the component being replaced
```

Example Utilization

```
ectf_replace -a /dev/ttyUSB0 -t 0123456789abcdef -i 0x11111126 -o 0x11111125
```

Attestation Tool

The attestation tool returns the confidential attestation data provisioned on a component. This is available on the PATH within the Poetry environment as `ectf_attestation`.

```
ectf_attestation --help
usage: eCTF Attestation Host Tool [-h] -a APPLICATION_PROCESSOR -p PIN -c
COMPONENT

Return the attestation data from a component

options:
  -h, --help            show this help message and exit
  -a APPLICATION_PROCESSOR, --application-processor APPLICATION_PROCESSOR
                        Serial device of the AP
  -p PIN, --pin PIN     PIN for the AP
  -c COMPONENT, --component COMPONENT
                        Component ID of the target component
```

Example Utilization

```
ectf_attestation -a /dev/ttyUSB0 -p 123456 -c 0x11111124
```