Florida Atlantic University

2024 MITRE ECTF DESIGN DOCUMENT

Team Members

Mila Anastasova
Rabih El Khatib
Zee Fisher
Angelina Guasti
Aimée Laclaustra
Daniel Owens
Reza Zilouchian

March 2024

Contents

1 Build Steps

1.1 Build Deployment

The deployment step is a process executed by the Host Computer and is responsible for generating data that will be included or referenced by the MISC device's firmware. The Build Deployment phase involves invoking the `ssl_gen_ca.sh` script in the Makefile to generate the following cryptographic materials:

- Root Certificate Authority (CA) key
- Root Certificate Authority (CA) certificate

The resulting root CA key and certificate ensures overall integrity and authenticity in the certificate chain.

1.2 Build Device

Following the generation of the root CA, the Build Phase focuses on the creation of cryptographic items for both the Application Processor and Component.

1.2.1 Application Processor

The Application Processor (AP) serves as the "master" of the medical device and manages communications with the Host Computer, coordinates required MISC functionalities, and performs the important processing tasks during the device's routine operations. Most importantly, the AP is tasked with maintaining integrity of the device.

During the Build Phase, the AP key and certificate are generated by invoking the `ssl_gen_device.sh` script and then subsequently signed by the root CA.

After the AP key and AP certificate have been generated, they are then copied to a header file (`secrets_ap.h`) with the root CA certificate for inclusion in the codebase.

The value of the token and pin chosen at build time are not known by the AP firmware. During the build process, hashes are generated of the pin and token using SHA2-512 and then are stored in the firmware. At runtime, when a pin or token needs validating, the SHA2-512 hash is calculated and compared against the hash stored in ROM.

1.2.2 Components

The Components embody the array of supplementary chips found in the medical device and operate under the governance of the AP, which they rely on to ensure the integrity of the device.

In the Build Phase, the Component key and certificate are generated alongside the AP key and certificate and signed by the root CA.

The root CA certificate, Component key, and Component certificate are then compiled to a header file (`secrets_component.h`) and stored for future use in the codebase.

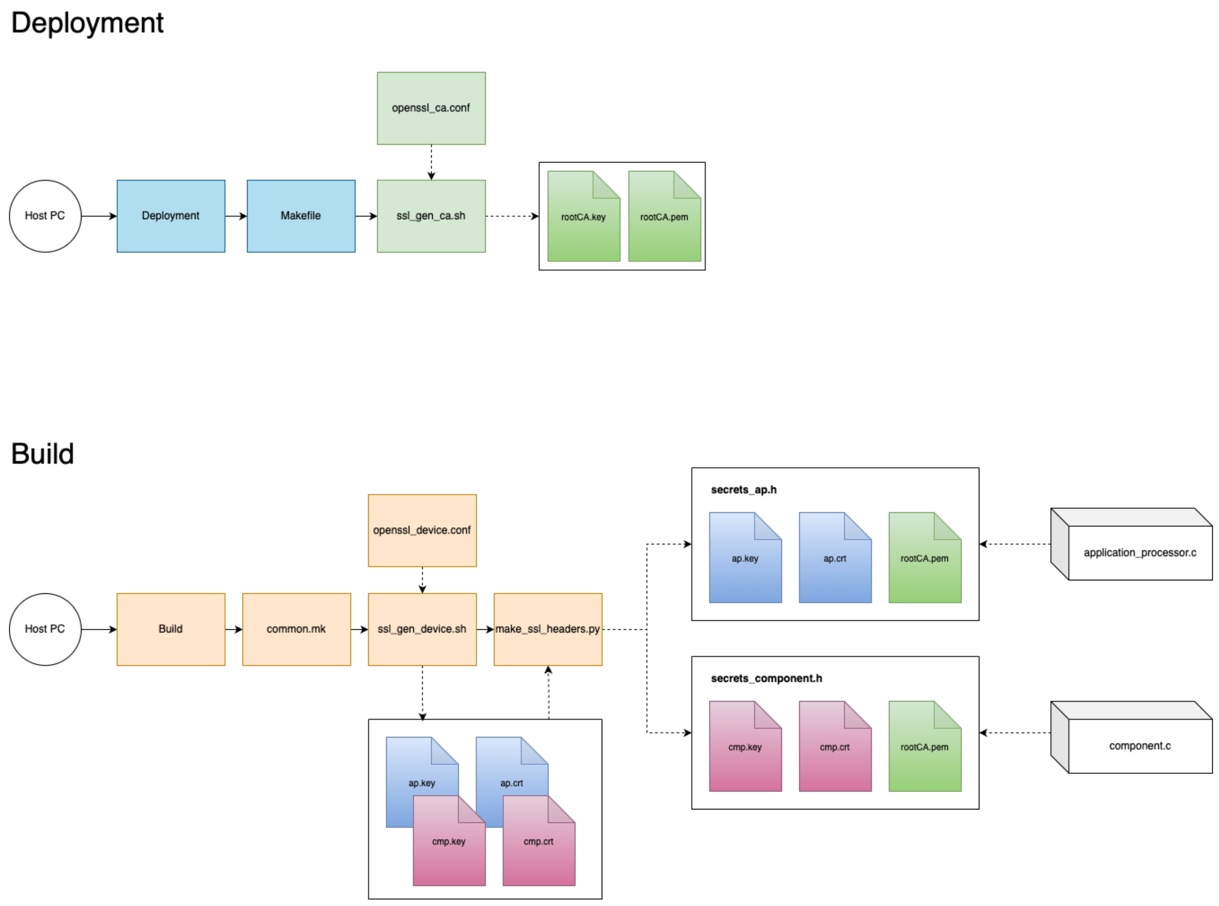

Figure 1.1: *Build phase*

Security Design

2.1 WolfSSL

For our security design, we utilize the wolfSSL library to implement Transport Layer Security (TLS) onto our medical device, providing robust and efficient cryptographic functionality without the need for custom-developed cryptographic solutions. wolfSSL's strong security and high performance are trusted and used by various industries, making it an ideal choice for our security design.

By using TLS, we ensure authentication, confidentiality, and integrity of data as it moves between devices. At a high level, the TLS protocol is used to secure communication in two steps:

1. **Asymmetric Key Exchange:** Establishes a secure session by generating a shared secret, leveraging the validity of asymmetric cryptography.
2. **Symmetric Encryption:** Ensures continuous protection with efficient symmetric encryption, maintaining the confidentiality and integrity of data in transit.

Initially, our strategy was to employ Ed25519 for the digital signature and Elliptic Curve Diffie-Hellman (ECDH) for key exchange. However, wolfSSL's implementation of Ed25519 was slower than anticipated and we were unable to enable optimizations due to space constraints.

As a result, we transitioned to using ECDSA (Elliptic Curve Digital Signature Algorithm) using the NIST curve P-256 instead. wolfSSL offers a highly optimized implementation of the P-256 elliptic curve, ensuring efficient cryptographic computations that align with our performance requirements. By using P-256 within wolfSSL, we were able to take advantage of the library's optimizations to improve the speed of operations.

2.1.1 I2C

The I2C functions provided by the organizers in the insecure firmware are limited to transmitting 255 bytes at a time. During handshake, wolfSSL needs to transmit in excess of this limit.

To address this, we wrapped the I2C send and receive functions in loops, checking against the expected number of bytes from wolfSSL and the total bytes sent or received to transmit data between devices in 255-byte chunks.

It seems possible to send more than 255 bytes in one packet, but exploratory modifications to the codebase did not work as hoped, so we decided to move forward with our original approach.

2.2 TRNG

Our security architecture utilizes the built-in hardware capabilities of the MAX78000FTHR board by enabling its True Random Number Generator (TRNG) with the following line of initialization:

C
```c
MXC_TRNG_Init();
```

Once initialized, we implement the board's TRNG as a reliable entropy source for wolfSSL's cryptographic operations to ensure robust randomness.

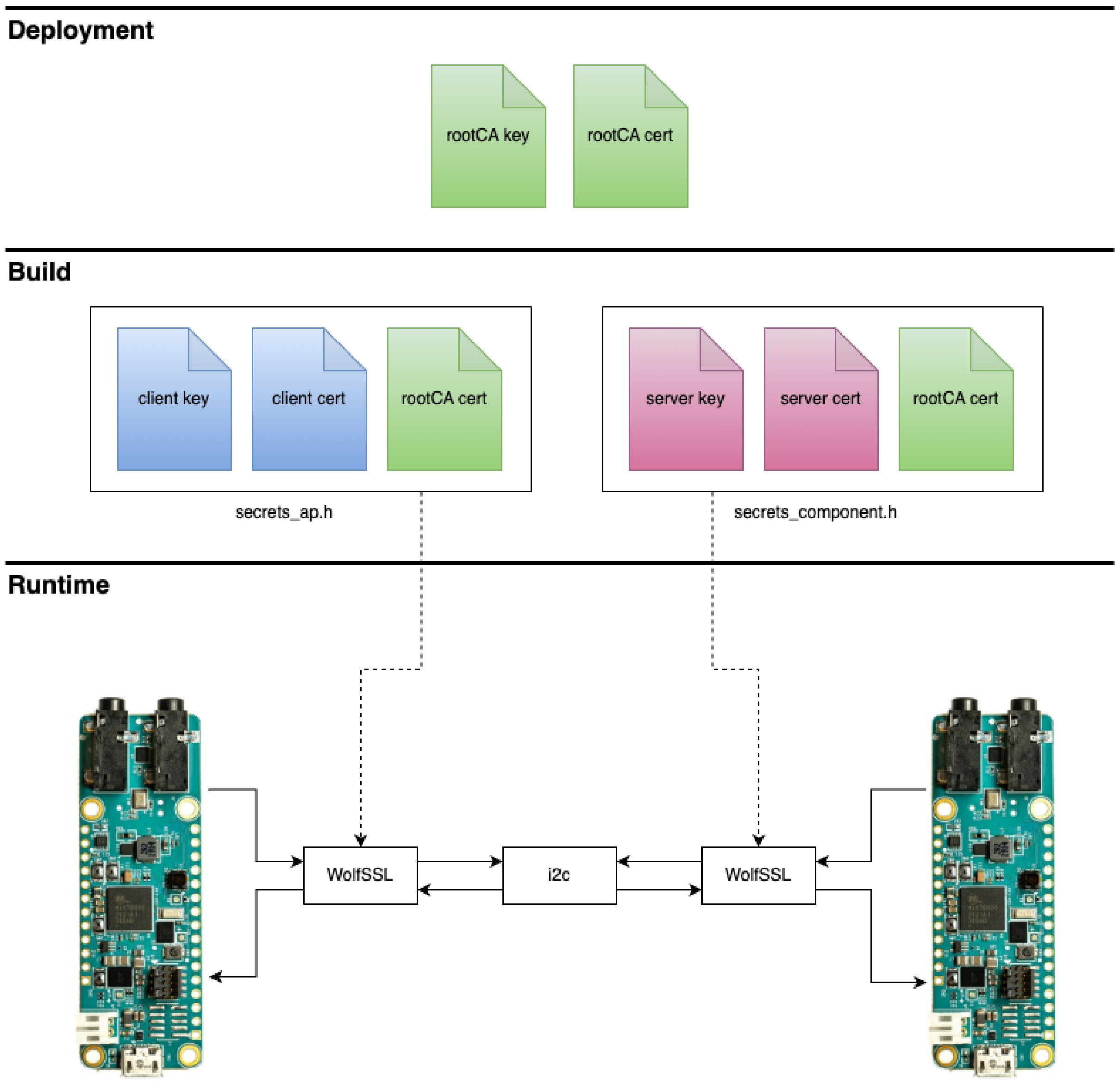

Figure 2.1: *Deployment of Certificates and wolfSSL to Secure Communication*

3 Defense

3.1 Compiler Counter Measures

In our defense strategy, we rely on the analysis provided by the GNU Compiler Collection (GCC) to enhance the security of our code. Hopefully, by doing so, we have modified our codebase to increase its resilience against potential attacks.

Specifically, we have applied the following compiler flags:

GCC
```
PROJ_CFLAGS += -D_FORTIFY_SOURCE=3 -fstack-protector-all
-mstack-protector-guard=global -Wformat -Wformat-security
-Werror=format-security
```

`D_FORTIFY_SORCE=3` compiler directive protects against buffer overflows by inserting checks to verify buffer boundaries on vulnerable operations.

We use `fstack-protector-all` so that a canary or guard variable is written at the beginning of each function. Upon the function's conclusion, this variable is inspected for any alterations, and if detected, it will the notify the environment.

`mstack-protector-guard=global` is another stack canary responsible for guarding the stack.

The use of `-Wformat -Wformat-security -Werror=format-security` flags turn format warnings into errors, thereby preventing security vulnerabilities that may arise from incorrectly formatted functions and variables.

We were able to compile wolfSSL using `D_FORTIFY_SOURCE=2`. However, adding additional flags resulted in an either excessive increase in the static library's size or unexpected side effects.

3.2 Vulnerable Functions

We have identified and replaced functions known for their vulnerability. Specifically, we transitioned from using unbounded string and input functions to their bounded, more secure counterparts to prevent buffer overflows.

Table 3.1: *Replaced Functions*

Vulnerable Function	Safer Alternative
`strcmp()`	`strncmp()`
`gets()`	`fgets()`

The gets() function is especially problematic since it will read an unbounded value into memory. We chose to use fgets() instead, which takes as input a maximum number of bytes to read.

In addition to changing the strcmp() function to strncmp(), we also ensured the buffers used for comparisons are set to a fixed sized corresponding to the length of incoming messages from the Host PC, a pin, or a token.

By transitioning to bounded alternatives, we hope to minimize the risk of buffer overflow attacks and improve the security of our codebase.

3.3 Disabling Unnecessary Hardware

To further reduce potential vulnerabilities, we have minimized the use of certain hardware components:

- Disabled the receive/transmit functionality on the audio jacks.
- Disabled the camera.

Other peripherals must be initialized before they are able to be shutdown or disabled; therefore, they are implied to be disabled by default.

By disabling unnecessary hardware, we aim to reduce the device's overall attack surface.

Florida Atlantic University

2024 MITRE eCTF Design Document

Mila Anastasova

Rabih El Khatib

Zee Fisher

Angelina Guasti

Aimée Laclaustra

Daniel Owens

Reza Zilouchian

March 2024