CyberSecurity Club at San Francisco State University

Initial Design Document

Ethan Hanlon[a], Michael Petrossian[a]

[a]San Francisco State University, 1600 Holloway Avenue, San Francisco, CA 94132, USA

Abstract

This research focuses on designing and implementing a secure MISC device that will be built to adhere to the standards of the functional and security requirements. Our design will aim to withstand all attack scenarios presented by the MITRE documentation, as well as provide a new approach to embedded system programming with the popular and recent memory-safe Rust programming language.

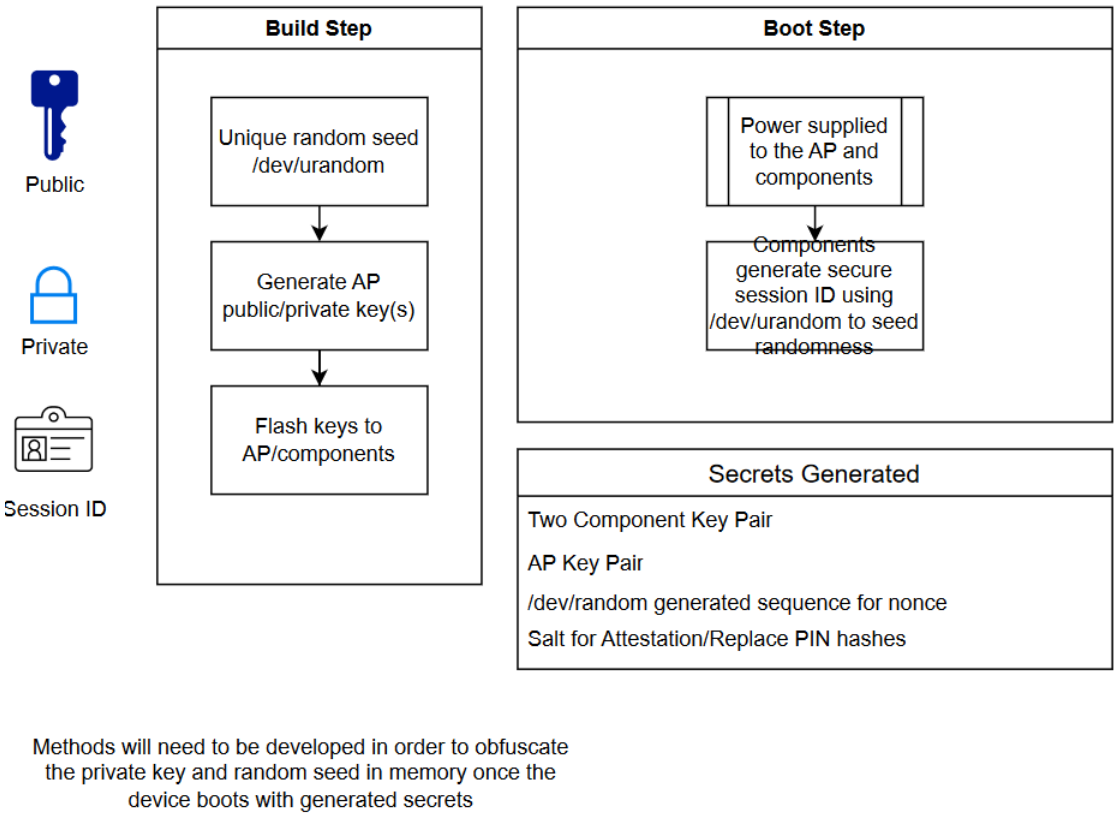
1. Introduction

The aim for this project is to create a device that can communicate securely and validate components for integrity and authenticity. In developing and implementing a MISC secure medical device, our team found that cryptography provided the strongest protection against most attack scenarios and fulfilled much of the requirements that demanded confidentiality, integrity, and authenticity. Although our design appears bullet-proof, there are still some fundemental flaws that, as will be discussed later, can only be mitigated rather than completely resolved. Our methods of encryption will comprise of elliptic curve key pairs, psuedo-random token seeds, and hash digests - all of which can be generated and derived from the build deployment.

2. Functional Requirements

2.1. Build Environment

The design was initially intended to be written in the Rust programming language for its consistency and safety guarantee. However, due to the lack of Rust support for the MAX78000 board and the complexities of intertwining it with the reference design's build system, sticking with C is the best option for this design.



2.2. Attest

When an authorized user enters the correct PIN, a unique sequence of numbers generated during the build deployment, they will be able to retrieve the attestation data through the AP.

2.3. Replace

An authorized user with a valid token can change the component ID that the AP is provisioning so long as the device is also provided a new "passcode" that will act as the new seed for the device's nonce to sync correctly (see section 3.1).

2.4. Boot

When the device boots, the AP and Components will verify each other in a process, described in section 3.1 and 3.2, where it will then be ready to communicate between each other for operation

2.5. Secure Communication

Secure communication will be perform through the I2C protocol. See section 3.5 for specifications.

3. Security Requirements

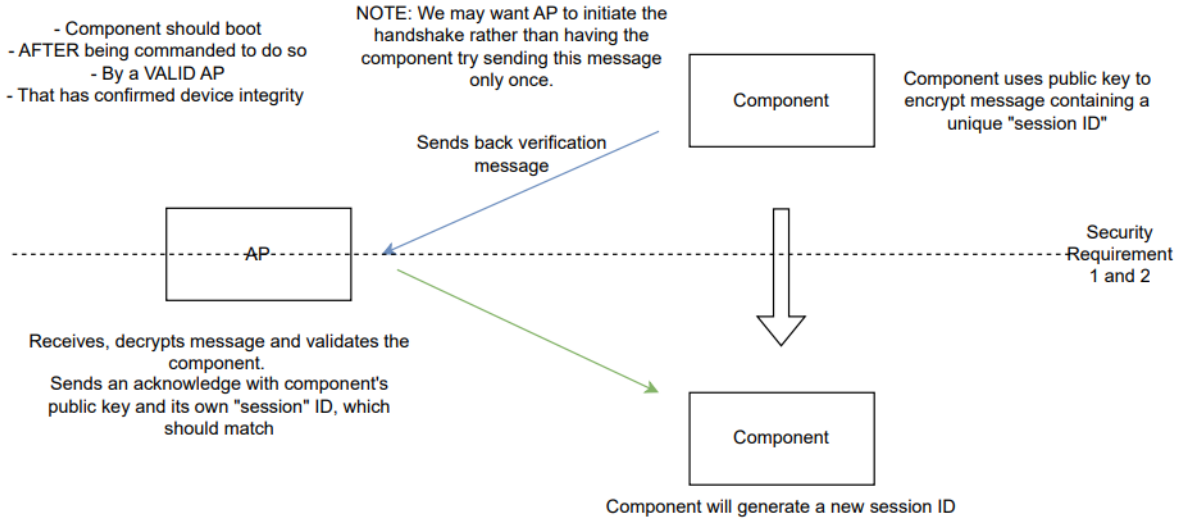
A key component of meeting the security requirements involves assymetric encryption and signature verification. To implement this, we will use the Elipctic Curve Cryptography (**ECC**) for its high security and low resource requirements, given the limited resources available to our embedded system. We will use the Curve25519 algorithm, which is widely considered to be secure and is known as one of the fastest ECC algorithms. For hashing, we will use the SHA256 algorithm, a highly secure and widely used hashing algorithm.

3.1. Security Requirement 1

Security Requirement 1 (**SR1**) requires us to ensure the Application Processor will not boot unless expected components are present and valid. Our plan for this involves a two-pronged approach which uses public/private key encryption and a secure cryptographic nonce.

During the build process, we will generate an ECC public/private key pair. The public key will be flashed onto the components, and the private key will be stored in the application processor. Additionally, a random seed will be generated using /dev/urandom and will be used to seed the random generator for both the application processor and component. This seed will be used by the components and the application processor to generate and validate cryptographic nonces - it will also prevent replay attacks from being performed in case the attacker sets up a MITM and tries to boot the AP using a valid signal but a malicious component.

The components will use the public key to encrypt the nonce along with their own unique identifier and send it back to the application processor over I2C bus. The application processor will use the private key to decrypt the nonce and verify that it matches the original nonce. If the nonces match, a new nonce is generated by the AP and sends out an acknowledgement to the component to continue the boot sequence. If no message is received, or the nonces do not match, the application processor will not boot.



3.2. Security Requirement 2

Security Requirement 2 (**SR2**) requires us to ensure that the components will not boot until commanded to do so by an Application Processor that has validated the integrity of the system. Our plan to meet this requirement is much the same as the plan for Security Requirement 1, but in reverse.

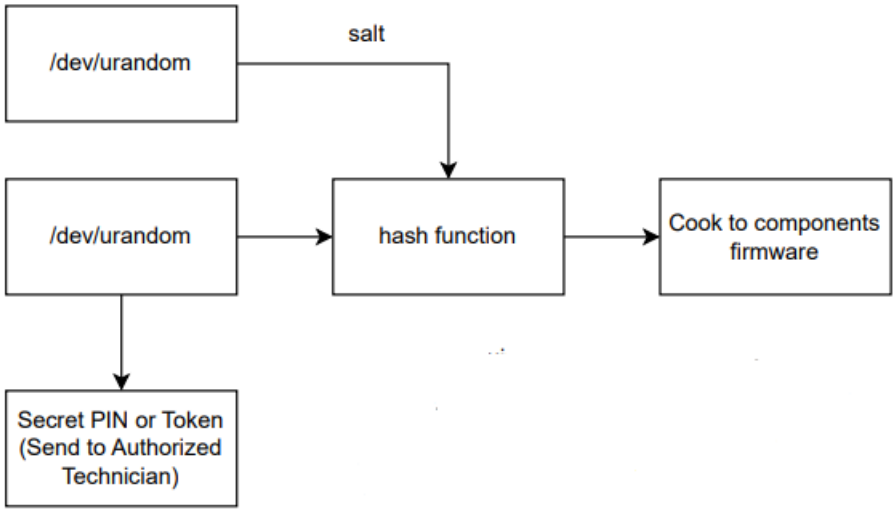
During the build process, a random seed will be burned onto the firmware as delineated in SR1. In addition, each component will receive an ECC public/private key pair similar to the AP's. This key will be used to encrypt messages before transmitting them over the I2C bus, as well as to verify digital signatures.

After the application processor verifies the message sent by the components as delineated in SR1, it will return an acknowledgement, encrypted with the component's public key. Upon receipt of this acknowledgement, the component will use its private key to decrypt the message and validate its digital signature. From there, the nonce will be checked against the random key generator. If the message is not present, the digital signature is invalid, or the nonce verification fails, the component will immediately halt the boot process. Conversely, if all checks pass - suggesting the AP successfully validated the components - the component will proceed with the boot sequence.

3.3. Security Requirement 3

Security Requirement 3 (**SR3**) Requires that the Attestation PINs and Replacement Tokens are kept confidential. To accomplish this, we will hash and salt the secrets to prevent the plaintext versions from being exfiltrated. It's unrealistic, given physical access to the machine, to expect that we can fully prevent the memory holding the secrets from being extracted from the device. However, by using SHA256, we can make the data useless to the attacker. By salting the PINs using a special token generated at compile time, we can prevent the use of a rainbow

table to get around the hashing. Only an authorized personal with the secret PIN is able to extract the attestation data.



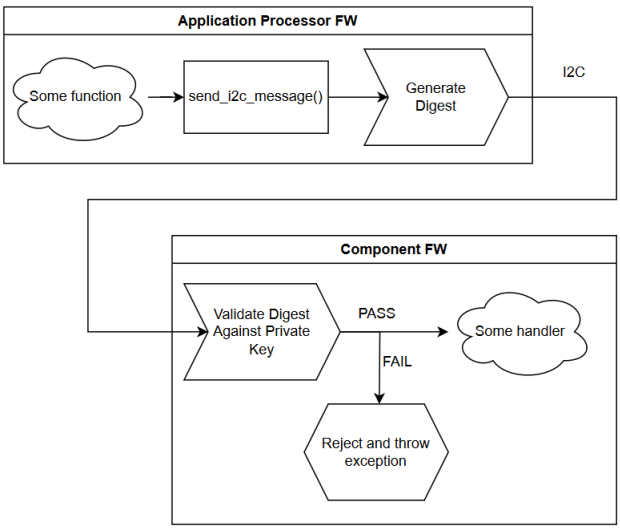
3.4. Security Requirement 4

Security requirement 4 (**SR4**) states that the attestation data is to be kept confidential and secure from unauthorized changes. Assuming that an attacker has the ability to modify the data directly, fulfilling this requirement can be done by encrypting the attestation data with the public key of the AP and embedding the data with its hash digest. Whenever an authorized user requests the attestation data, the AP first decrypts the data received by the component and then verifies the integrity by matching the resulting hash of the data with the embedded digest that went along with it. This technique is inspired by modern message authentication with message authentication codes (MAC) ensuring the integrity of data whenever it is in transit.

Complexities arise when integrating the encryption scheme into the build system of the MITRE reference design that this device relies on. Due to technical difficulties, the attestation data cannot be encrypted at compile time, so the moment the component boots, it will encrypt the plaintext attestation data at runtime.

3.5. Security Requirement 5

Security requirement five (**SR5**) requires us to ensure the integrity and authenticity of all post-boot MISC secure communications. Our plan to solve this challenge is to write validation requirements directly into the firmware. All messages bound for the I2C bus will have a digest created based on the device's private key, which was generated on the build step for each device for SR1 and SR2. When a message is received, the firmware will first check the message against the digest. If the message fails to validate, we can assume it is either damaged or inauthentic and reject it accordingly.



Acknowledgements

This work was supported by the students and faculty of the CyberSecurity Club at San Francisco State University. We would like to thank the club for their support and guidance.