eCTF : Final Design Document

Virginia Tech - Embedded Capture the Flag Team:
Rohit Mehta, Rohit Sathye, Wesley Flynn, Raghav Agarwal

April 5, 2024

Abstract

The three main functions of the system are “Boot”, “Attest” and “Replace”. We have structured our design considering the CIA Triad : Confidentiality, Integrity, and Availability of the various components in these functions.

Security Requirements

- Security Requirement 1:
  The Application Processor (AP) should only boot if all expected Components are present and valid. If the Medical Device is not in a fully functional state, the AP should not boot. If the AP is able to turn on in an invalid state, patient health and data could be at risk. The Medical Device’s AP should confirm the device’s integrity before booting.

- Security Requirement 2:
  Components should only boot after being commanded to by a valid AP that has confirmed the integrity of the device. Any Component present on a Medical Device should only boot if it is part of a valid system. If a Component is able to be used outside of a valid device – including on counterfeit devices – the safety of patients and the reputation of the company could be at risk. Sensitive data on the Components may also be at risk to being leaked if the Component is able to boot. As the AP oversees validating the device integrity, the Components should not boot until told to do so by a valid AP.

- Security Requirement 3:
  The Attestation PIN and Replacement Token should be kept confidential. Attestation PINs (see Attest) and Replacement Tokens (see Replace) play an integral role in segmenting access to privileged operations in Medical Devices. Any number of actors could gain access to a functioning device, so it is crucial that they are not able to extract these secrets from it. If someone were able to, they would be able to gain access to sensitive medical data and to introduce counterfeit parts into otherwise legitimate systems.

- 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. The attestation data on each Component is essential to determining the validity of the Component. If an attacker is able to access this data without the required privilege, they may be able to recreate or modify the critical Component. A leak of sensitive propriatary information would damage your companies reputation and could potentially lead to counterfeit Components risking patient safety.

- Security Requirement 5:
  The integrity and authenticity of messages sent and received using the post-boot MISC secure communications functionality should be ensured. The ICs that your manufacturer creates will end up serving critical roles in Medical Devices. Once the system is active, the communications between the components must be secure from tampering, duplication, or forgery. If not, patient safety could be at risk to malicious attacks or incidental data corruptions.

Secure Boot

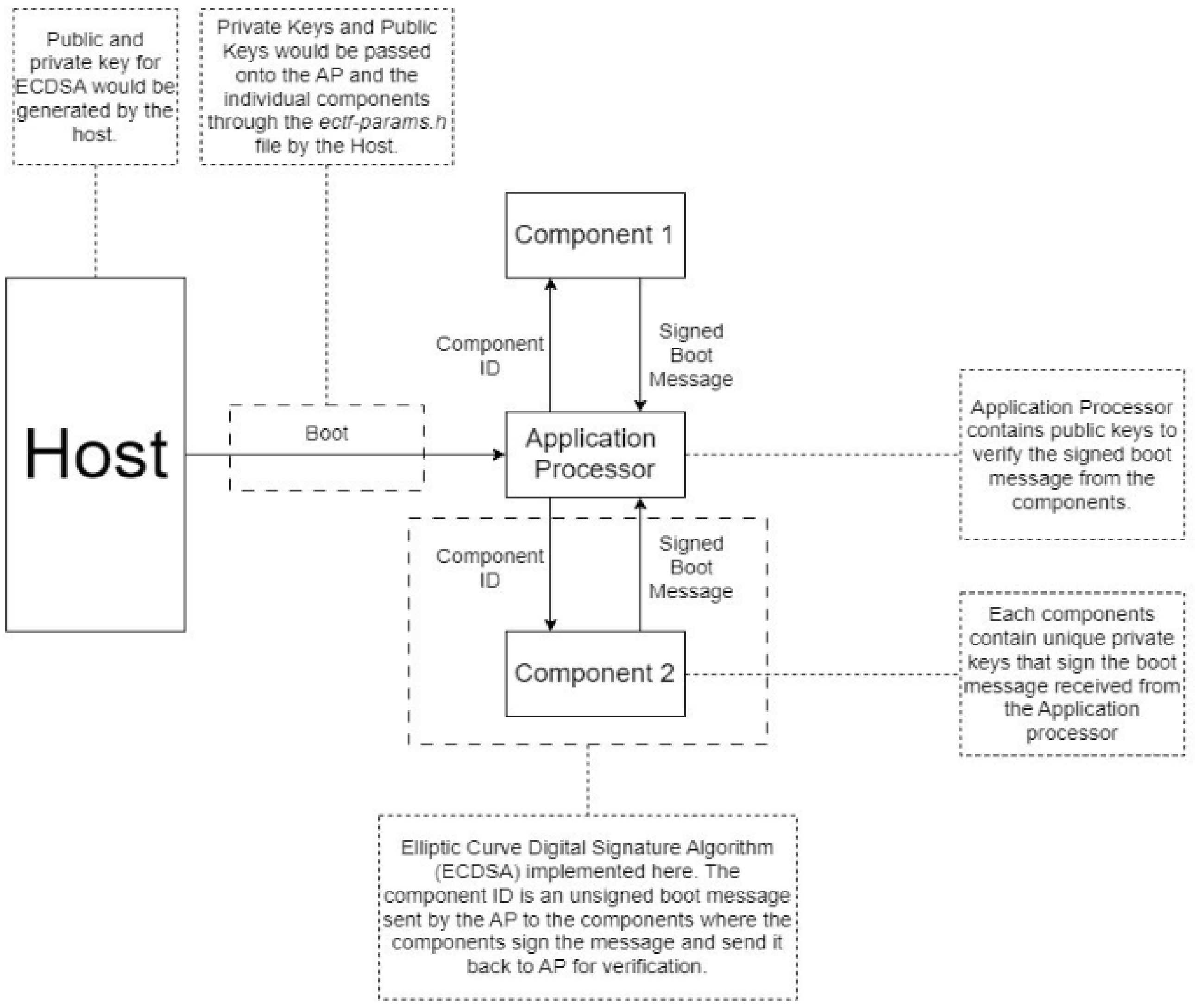
Public and private key for ECDSA would be generated by the host.
Private Keys and Public Keys would be passed onto the AP and the individual components through the ectf-params.h file by the Host.
Component 1
Component ID
Signed Boot Message
Host
Boot
Application Processor
Application Processor contains public keys to verify the signed boot message from the components.
Component ID
Signed Boot Message
Component 2
Each components contain unique private keys that sign the boot message received from the Application processor
Elliptic Curve Digital Signature Algorithm (ECDSA) implemented here. The component ID is an unsigned boot message sent by the AP to the components where the components sign the message and send it back to AP for verification.

Figure 1: Secure Boot Flowchart

This funcitonalty is implemented as per the initial desgin document. To implement the boot functionality for a Medical Device (MISC), a robust and secure process must be established. The solution to offering security to this scenario is through Authentication. We propose using the Elliptic Curve Digital Signature Algorithm (ECDSA) to enhance the security of the boot process and prevent the spoofing of component IDs as a part of the authentication process.

Initially public and private key pairs would be generated by the host and stored in a python file. In the next step it is essential to supply these keys to the elements. Thus, during the deployment phase, public and private key pairs would be distributed to the application processor and the components. Thereafter, the design flow is like authenticating a digital signature. This is done using a Python file which is included in the deployment folder.

The python file implements the Elliptic Curve Digital Signature Algorithm (ECDSA) to generate cryptographic key pairs for both an access point (AP) and a client. Utilizing the ECDSA, the code creates a private key for each entity and derives their corresponding public keys. These keys are crucial components in cryptographic operations, serving to authenticate and secure communication between the access point and the client.

Once the key pairs are generated, the code proceeds to store them in separate header files, namely AP_Keys.h and Comp_Keys.h. The private keys are written as hexadecimal strings directly into these files, facilitating easy retrieval and usage in subsequent cryptographic processes. Additionally, the code calculates the X and Y coordinates of the public keys and ensures their uniform length by padding them with leading zeros if necessary.

These header files serve as repositories for the cryptographic keys, allowing other components of the system to access and utilize them as needed. By separating the keys into distinct files, the code promotes modularity and organization within the cryptographic system, enhancing its maintainability and scalability.

Furthermore, the generated key pairs enable secure communication between the access point and the client by enabling cryptographic operations such as digital signature generation and verification. Through ECDSA, the keys provide a robust mechanism for ensuring data integrity and authenticity, safeguarding sensitive information exchanged between the two entities.

Overall, the code encapsulates the generation and storage of cryptographic key pairs using ECDSA, providing a foundational element for establishing secure communication channels in networked systems. Through its modular approach and utilization of standardized cryptographic techniques, the code contributes to the creation of resilient and trustworthy communication infrastructures.

Furthermore the code directory has another folder named Shared, where signing, verification, hash and TRNG generation code is present. Now we'll discuss these functions in more detail.

- Signing: The sign_key function our design is a crucial component for generating cryptographic signatures using a private key, a hash message, and a specified elliptic curve. It utilizes dynamic memory allocation to create space for the resulting signature, allocating memory for 64 bytes to accommodate the signature's size. The private key is extracted from the input string and converted into bytes, stored in an array for subsequent use. By iterating over the private key string and converting pairs of characters into hexadecimal values, the function prepares the necessary data for the signing process. Once the private key is prepared, the function invokes the uECC_sign function, likely provided by a cryptographic library, to perform the signing operation. If the signing process is successful, denoted by the uECC_sign function returning a value of 1, the function returns a pointer to the generated signature. However, in the event of a failure during the signing process, the allocated memory for the signature is freed to prevent memory leaks, and the function returns NULL. While additional functionality for error handling and message printing is present in commented-out lines, its current omission suggests a focus on core functionality.
  *It should be noted that after the usage of signature the allocated memory should be freed using the free() function to avoid memory leaks.

- Verification: The verify_key function is designed to verify the authenticity of a public key using elliptic curve cryptography (ECC). It takes several parameters including the x and y coordinates of the public key, the hash value, the signature, and information about the ECC curve. The function first constructs an array called public_key to store the concatenated x and y coordinates of the public key. It then iterates over the hexadecimal strings representing the x and y coordinates, converting them into decimal values and storing them in the public_key array.
  Following the construction of the public_key array, the function calls the uECC_verify function to verify the authenticity of the key. This function utilizes the ECC curve information along with the

public key, hash value, and signature to perform the verification process. Upon completion, the uECC_verify function returns an integer value indicating the result of the verification. If the result is equal to 1, the function returns a constant value SUCCESS_RETURN, indicating a successful verification of the key's authenticity. However, if the verification result differs from 1, the function returns ERROR_RETURN" after introducing a delay of 1000 milliseconds.
In summary, the verify_key function plays a crucial role in confirming the validity of a public key through ECC mechanisms. By converting the provided hexadecimal coordinates into decimal values and utilizing ECC-based verification processes, the function ensures the integrity and authenticity of the key. Additionally, the function incorporates error handling measures, returning specific constants to indicate the outcome of the verification process and introducing a delay in case of errors.

- Nonce & Hash Generation: The generate_nonce function defined in the RNG.c file serves the crucial purpose of producing a random nonce, essential for cryptographic operations. Upon invocation, this function initializes an array named "nonce" to store the generated nonce, followed by allocating memory for another array named message_hash to hold the hash of the generated nonce. The initialization process ensures that both arrays are properly initialized and ready for use in subsequent steps.
  Subsequently, the function activates the True Random Number Generator (TRNG) module via the MXC_TRNG_Init() function, facilitating the generation of random bytes to populate the nonce array. Following the generation of the random nonce, the TRNG module is shut down using the MXC_TRNG_Shutdown() function, maintaining system resources efficiently.
  Moreover, the function employs the SHA-256 hashing algorithm to compute the hash of the generated nonce, enhancing the nonce's security and integrity. By initializing the hashing algorithm using sha256_begin() and invoking sha256() to compute the hash, the resulting hash value is stored in the message_hash array. Finally, the function returns the computed hash, which can be utilized in cryptographic processes requiring a secure and random nonce.

Now we'll see how the application processor and component validate each other and the secured boot functionality is carried out to fulfill the security requirement 1. Let's see what happens in the application processor, the validate_components() function in the AP serves a crucial role in validating components within a system by orchestrating communication with each component and assessing their validation results. To facilitate this process, the function begins by declaring three arrays: receive_buffer, transmit_buffer, and rng_nonce, which are utilized for communication and data storage during validation operations. Initialization of a random number generator via the RNG_init() function with the rng_nonce array as the seed ensures the unpredictability of generated random numbers, vital for cryptographic operations.

Subsequently, the function configures the elliptic curve cryptography library by invoking the uECC_set _rng() function with the RNG_init function as the argument, thus indicating the utilization of random numbers generated by RNG_init for cryptographic purposes. Additionally, parameters for the secp256k1 elliptic curve, commonly employed in cryptographic operations, are retrieved through the uECC_secp256k1() function. The function then iterates over each component in the system, determined by the variable flash_status.component_cnt, to perform individual validation checks.

Within the loop, the function communicates with each component by sending a validation command and subsequently receiving validation responses. The received signature and hash values are then verified using the component's public key and curve parameters. Further operations involve generating a nonce, signing it with the application processor's private key, and transmitting the signed package to the component for validation. The function meticulously handles errors throughout the validation process, ensuring proper error reporting and termination of validation operations in case of failure. Upon completion of validation checks for all components, the function returns SUCCESS_RETURN, signaling the successful validation of all system components. The presence of commented-out lines indicates potential provisions for debugging or logging, offering insights into the function's execution but currently disabled for operational purposes.

On the component, the function named validate_components(), conducts a systematic validation of components within a system. It begins by initializing communication buffers and the random number generator. Subsequently, it iterates through each component in the system, sending validation commands and receiving responses. Within this loop, it ensures proper communication with each component, verifies received cryptographic signatures against expected hashes, and generates and sends its own signatures for further validation. This process employs elliptic curve cryptography techniques to ensure the integrity and authenticity of each component.

Throughout the validation process, the function meticulously manages error handling, printing error messages if communication failures or verification issues arise. It dynamically allocates memory for temporary storage of cryptographic data and efficiently frees it after each step. Additionally, it cross-validates received validation results against expected component IDs to ensure consistency and accuracy. Ultimately, the function provides a robust mechanism for ensuring the integrity and security of system components, crucial for maintaining overall system reliability and trustworthiness.

Secure Attest

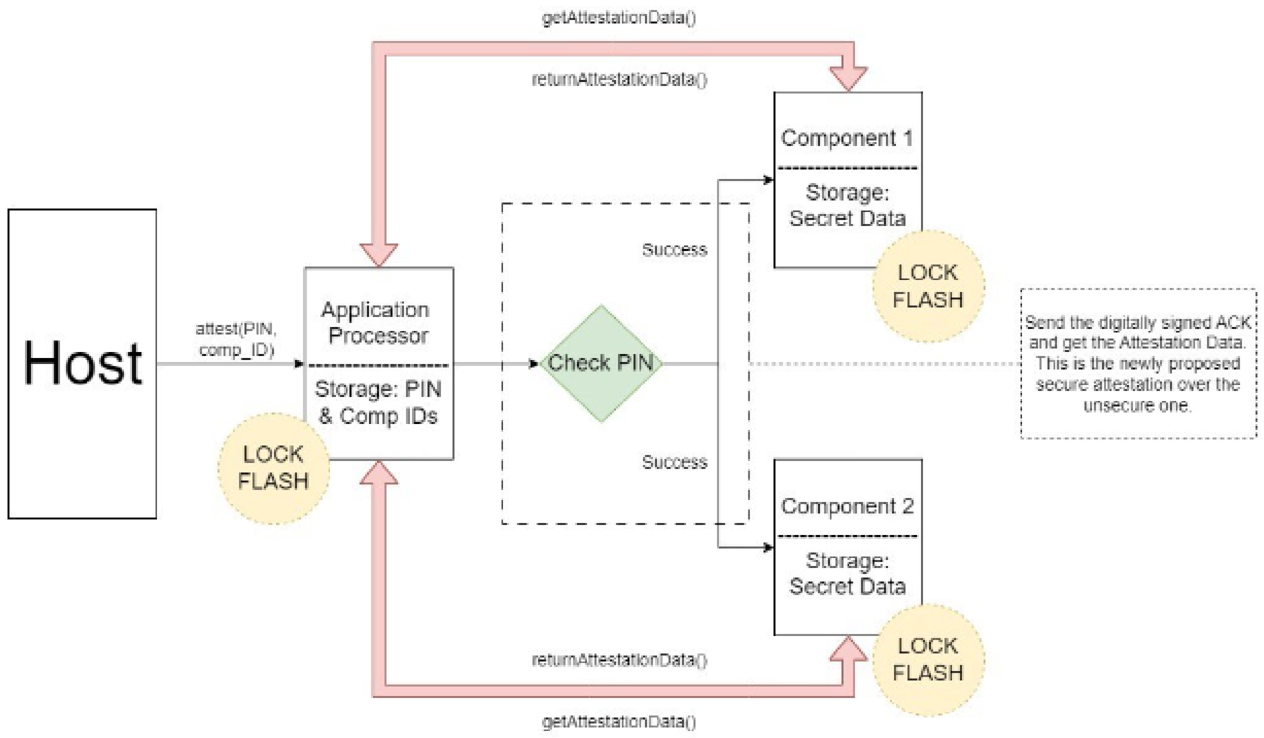

Figure 2: Secure Attest Flowchart

The attestation process consists of the following steps:

1. The user enters the PIN, and the AP verifies the PIN
2. AP queries the component for the attestation data.
3. Component responds with the attestation data.

The problem in this scenario is that the component never knows if the AP has successfully verified the PIN. If a malicious AP was to issue the getAttestationData command without verifying the PIN, the component would return the data.

The secure attest works similar to the validate component function of the secure boot. Both the application processor and the component validate each other, and once the component verifies that it's the valid application processor, it sends all the sensitive data.

Secure Replace

The system's component replacement procedure involves a user authentication step followed by the input of the IDs of both the new and old components. Only the validation of the user's token requires stringent security measures. Following successful token validation, the user can proceed to input the relevant component IDs. As the component IDs themselves are not considered sensitive information, the focus of security efforts primarily lies in authenticating the user through their token.

However, due to constraints such as design complexity and time limitations, complete development of the replacement function was not feasible. Consequently, while the authentication process is securely implemented to validate the legitimacy of the user, the actual replacement functionality remains incomplete. This incomplete development may encompass various aspects, including but not limited to error handling, data validation, and the integration of component replacement logic. Despite this, the fundamental framework for securely authenticating users and initiating the replacement process is established, albeit requiring further development to achieve full functionality.