Medical Device (MISC) Design for MITRE eCTF 2024

SIGPwny at the University of Illinois Urbana-Champaign (UIUC)

1 Overview

This Design Document describes UIUC's implementation of the MISC system as defined by MITRE for eCTF 2024. The MISC system builds the platform for secure communication between medical devices through the use of a central Application Processor (AP) and various Components. While the reference design is considered insecure, our implementation aims to provide a secure and robust system that meets the security requirements outlined by MITRE.

We started our design by analyzing the security requirements and creating the conceptual protocols and cryptographic schemes that would meet these requirements.

1.1 Rust

Our design is built using Rust, a systems programming language that provides memory safety, a strong type system, and a minimal runtime. Crucially, Rust protects us from common security vulnerabilities such as buffer overflows. That being said, Rust is not a silver bullet, and we must still be careful to avoid other security pitfalls.

There were quite a few challenges related to using Rust. In particular, the embedded platform that we are building on (Analog Devices MAX78000FTHR) did not have a stable, public Rust toolchain available online. To solve this, we implemented the peripheral drivers and built our own Hardware Abstraction Layer (HAL) from scratch.

Another challenge was that we were required to support the execution of post boot code which is written in C. Not only that, but the C code needed to be able to call back into our Rust code in order to use our secure communication layer. This post boot code also requires the availability of functions from both the standard C library and the MSDK. We rewrote these functions in Rust and provided a C ABI for them so the linker could provide them to the post boot C code.

2 HIDE Protocol Communication Layer

We implement an extra communication layer between the I2C layer and the application layer, which we refer to as the HIDE protocol. The HIDE protocol ensures that all messages maintain confidentiality, integrity, authenticity, and non-replayability. We require all messages sent between the AP and the Component to use HIDE.

HIDE effectively turns each application message into a three-way challenge-response handshake. The sender first initiates a message request. The receiver will then send a random, encrypted challenge. The sender will then decrypt the challenge, solve it, and encrypt the challenge response to be sent along with the actual message. To solve the challenge nonce, the sender must perform a bitwise XOR of `0x55` with each byte in the challenge nonce.

We use the Authenticated Encryption (AE) cipher, Ascon-128, for our cryptographic scheme. We chose Ascon since it was selected in the NIST Lightweight Cryptography competition and has a masked software implementation that has been tested against various power analysis and hardware attacks.

We use Ascon's associated data feature to validate each message that uses encryption. The associated data is 8 bytes long, with the first 4 bytes being the component ID, the fifth byte being the HIDE message magic byte, and the last three bytes being null bytes.

Each direction of communication uses a different symmetric encryption key, meaning there are two encryption keys:

- $K_{AP,C}$ is the key for messages sent from the Application Processor to the Component.
- $K_{C,AP}$ is the key for messages sent from the Component to the AP.

Every AP and Component built from the same deployment will share the same keys. This allows any Component to communicate with an AP from the same deployment.

2.1 HIDE Secure Protocol

If at any point the encryption or decryption of a HIDE message using the Ascon cipher fails (such as invalidation due to message bit-flips or incorrect associated data), our secure communication functions will return an error.

Messages sent using HIDE can have a length up to 80 bytes. When HIDE interfaces are being used in the post boot code, the message length is limited to 64 bytes and the message length is encoded as an 8-bit integer in the first byte of the message. The remaining bytes are padded with random bytes. MISC messages are also limited to 64 bytes in length.

2.1.1 HIDE_PKT_REQ_SECURE

Sent by any device to initial a secure communication session with another device. This packet is unencrypted and contains the magic bytes corresponding to a secure request.

Name	Offset	Size (bytes)	Content
Magic	`0x00`	10	`\x40` * 10

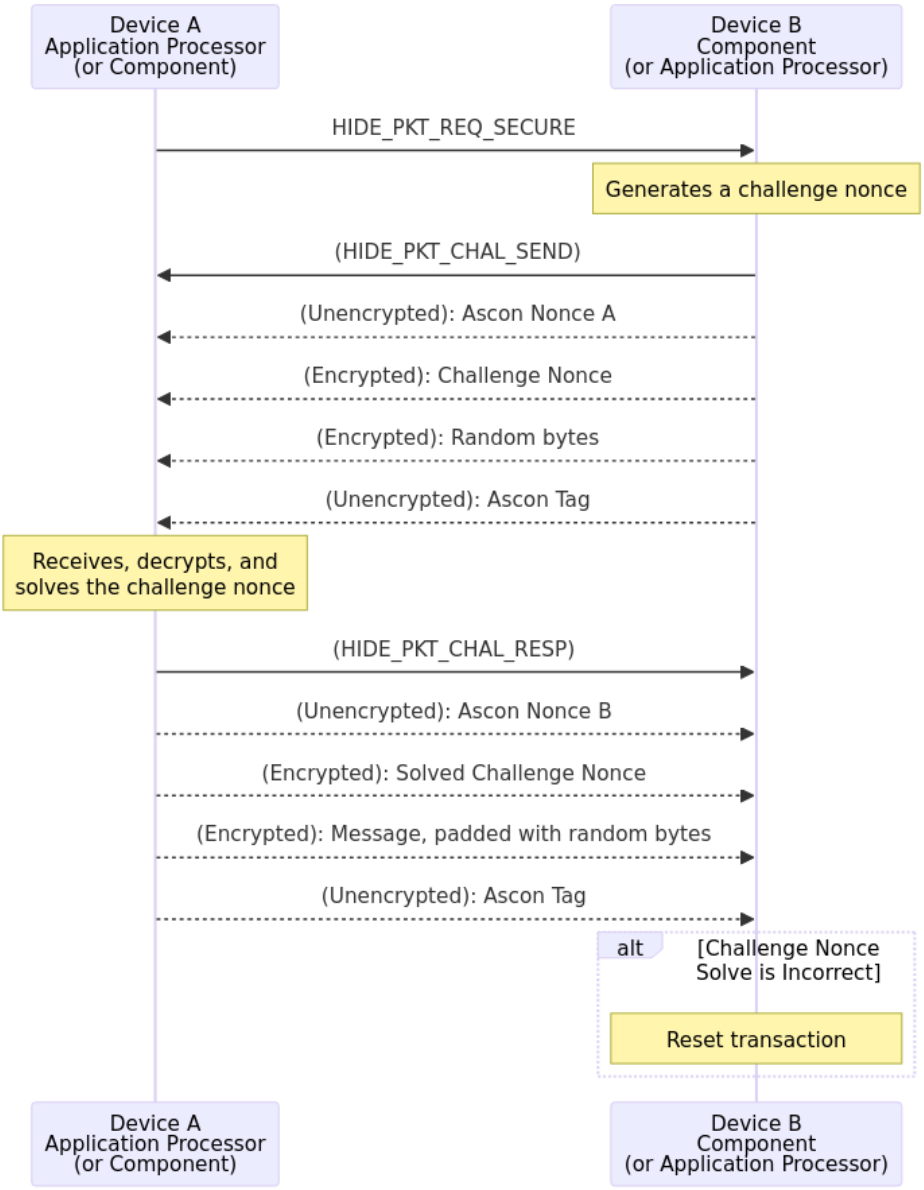

Figure 1: HIDE secure communications protocol

2.1.2 HIDE_PKT_CHAL_SEND

Sent by the receiver of a secure request to initiate the challenge-response handshake. This packet contains the Ascon Nonce, the encrypted challenge nonce, and random bytes for padding.

Name	Offset	Size (bytes)	Content
Ascon Nonce	`0x00`	16	`\x?? * 16`
Encrypted Challenge Nonce	`0x10`	16	`\x?? * 16`
Encrypted Random Bytes	`0x20`	80	`\x?? * 80`
Ascon Tag	`0x70`	16	`\x?? * 16`

> [!WARNING] Ascon Nonce should be randomly uniquely generated for all messages

2.1.3 CHAL_RESP

Sent by the sender of a secure request to complete the challenge-response handshake. This packet contains the Ascon Nonce, the encrypted solved challenge nonce, and the message to be sent.

Name	Offset	Size (bytes)	Content
Ascon Nonce	`0x00`	16	`\x?? * 16`
Encrypted Solved Nonce	`0x10`	16	`\x?? * 16`
Encrypted Message + Random	`0x20`	80	`\x?? * 80`
Ascon Tag	`0x70`	16	`\x?? * 16`

> [!WARNING] Ascon Nonce should be randomly uniquely generated for all messages

2.2 HIDE List Protocol

As part of the MISC system, we also must implement functionality to list all Components connected to the AP. We would use the HIDE secure protocol to enumerate I2C addresses and discover components, however, the HIDE secure protocol uses a Component's full ID as associated data during the encryption and decryption process. During this discovery process, we do not know the full ID of any Component with the exception of the Components which are provisioned for the AP. Even then, we cannot use these provisioned Component IDs in the case that a different valid Component shares the same I2C address. So, we decided to implement an extension to the HIDE protocol, which we refer to as the HIDE list protocol. This is only supported when the AP is communicating with a Component.

The AP will send an unencrypted request packet which contains the magic bytes corresponding to a list request. The Component will then respond immediately with its 4-byte Component ID.

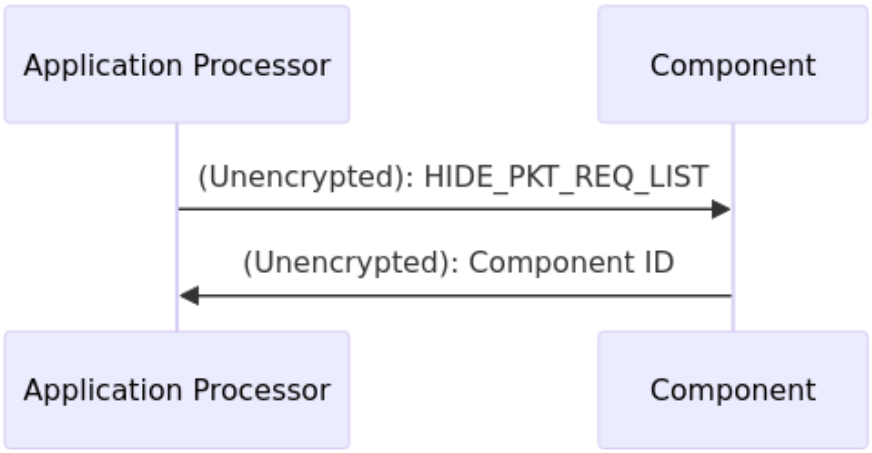

Figure 2: HIDE list protocol

3 MISC Protocol

The Medical Infrastructure Supply Chain (MISC) defines the functionality of our medical device design. We formerly describe protocols for each function in MISC. In order to maintain the secure operation of devices, all MISC messages are sent using the HIDE protocol.

> [!NOTE]
> "TTT" refers to "total transaction time" and is used to ensure timing functionality requirements are met.

3.1 List Components

The AP must be able to list Components which are currently connected to it, regardless of whether a Component has been provisioned for the AP. This is done by sending a HIDE list packet request to each possible I2C address. If a Component is present, it will respond with its Component ID. The AP will then log this Component ID to the host.

Since listing a component is not considered a sensitive operation, there is no requirement to use secure communication. Additionally, performing a full HIDE challenge-response handshake for every possible I2C address would be time-consuming and unnecessary. The AP would also need to know the Component's ID in order to properly encrypt/decrypt a secure HIDE message. For these reasons, the HIDE list packet request was created as an extension to the HIDE protocol and allows the quick retrieval of a Component's ID without the need for secure communication.

Note that the listing components protocol is an exception to the rest of the MISC protocol, which use the full HIDE protocol by sending a HIDE secure packet request.

Specific message packet formats are not defined for the list components protocol, since this is defined by the HIDE protocol.

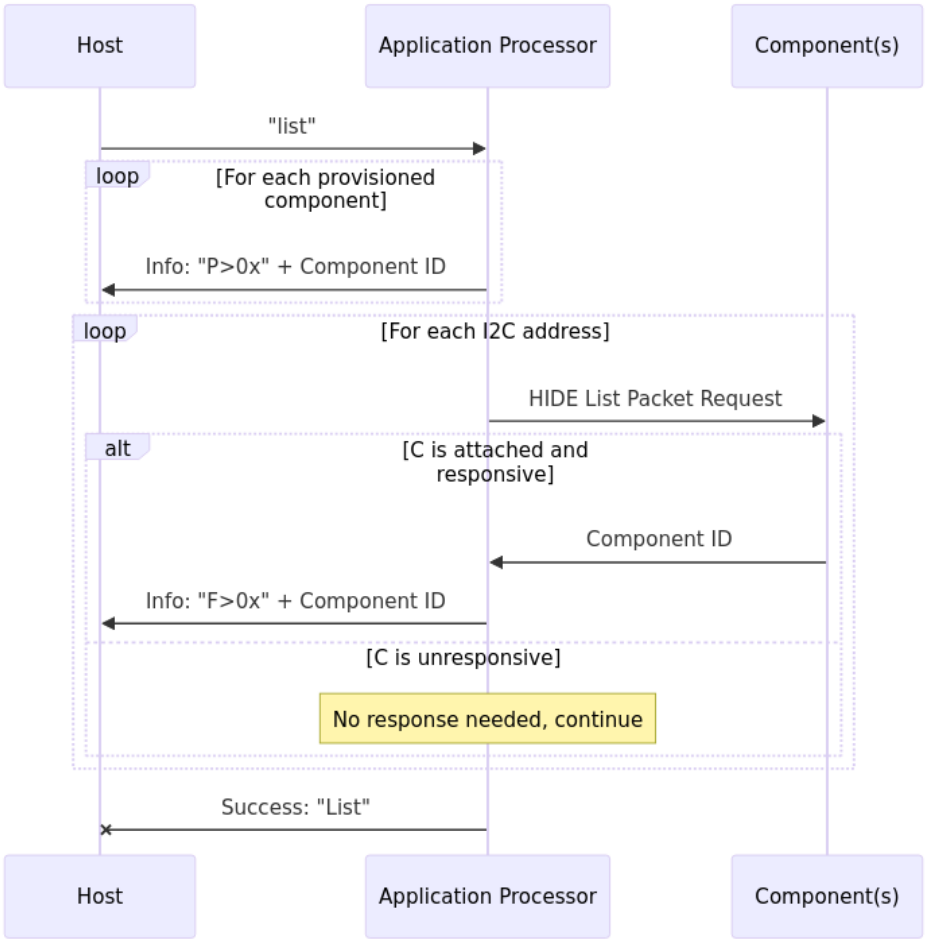

Figure 3: List components protocol

3.2 Attest Components

Every Component stores Attestation Data which is used to verify the authenticity of the Component. This Attestation Data is protected by a 6-byte Attestation PIN, which must be stored and validated on an AP. Thus, the communications between the AP and the Component must be authentic and secure in order to prevent possible attacks to extract the Attestation Data through bypassing the AP. The HIDE protocol is used to ensure this secure communication.

In order to protect the Attestation PIN from being compromised, the AP stores the Attestation PIN as three uniquely salted SHA3-512 hashes. The salt is a random 16-byte string that is generated during the AP's build process and is prepended to the Attestation PIN before hashing. Each salt and hash pair is stored in the AP so that the Attestation PIN is never stored in plaintext.

The transfer of the Attestation Data is broken into three parts in order to meet length requirements of the HIDE protocol. The AP will first request the Component's Attestation Location, then the Attestation Date, and finally the Attestation Customer. The Component will respond with each of these pieces of information in turn. Only once the AP has received all three pieces of information will it send the Attestation Data to the host.

Additionally, a fixed transaction delay of 2.5 seconds is sent in place to prevent brute force attacks.

3.2.1 REQUEST_LOCATION

The Application Processor sends this message to the Component to request the Component's Attestation Location. The Component will respond with its Attestation Location.

Name	Offset	Size (bytes)	Content
Magic	`0x00`	80	`\x60` * 80

3.2.2 SEND_LOCATION

The Component sends this message to the Application Processor in response to a REQUEST_LOCATION. This message only contains the Component's Attestation Location.

Name	Offset	Size (bytes)	Content
Location	`0x00`	64	`\x?? * 64`

3.2.3 REQUEST_DATE

The Application Processor sends this message to the Component to request the Component's Attestation Date. The Component will respond with its Attestation Date.

Name	Offset	Size (bytes)	Content
Magic	`0x00`	80	`\x62` * 80

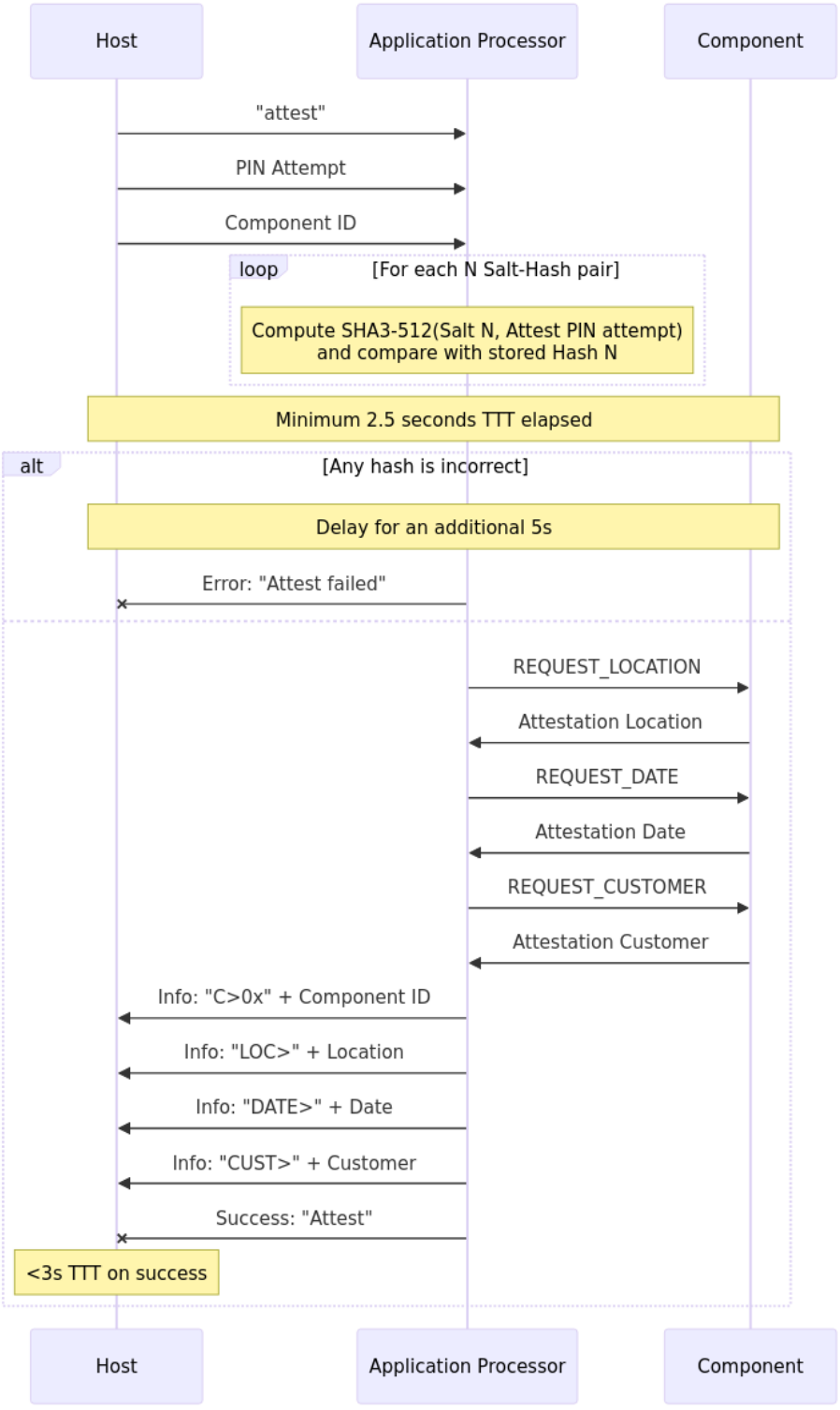

Figure 4: Attest components protocol

[!WARNING]
The Component should not respond to a REQUEST_DATE if it has not received a REQUEST_LOCATION beforehand.

3.2.4 SEND_DATE

The Component sends this message to the Application Processor in response to a REQUEST_DATE. This message only contains the Component's Attestation Date.

Name	Offset	Size (bytes)	Content
Date	`0x00`	64	`\x?? * 64`

3.2.5 REQUEST_CUSTOMER

The Application Processor sends this message to the Component to request the Component's Attestation Customer. The Component will respond with its Attestation Customer.

Name	Offset	Size (bytes)	Content
Magic	`0x00`	80	`\x64` * 80

3.2.6 SEND_CUSTOMER

The Component sends this message to the Application Processor in response to a REQUEST_CUSTOMER. This message only contains the Component's Attestation Customer.

Name	Offset	Size (bytes)	Content
Customer	`0x00`	64	`\x?? * 64`

[!WARNING]
The Component should not respond to a REQUEST_CUSTOMER if it has not received a REQUEST_DATE beforehand.

3.3 Replace Components

The AP keeps tracks of Components which are provisioned for it by storing their Component IDs. When a Component is replaced, the AP will update its list of provisioned Components with the new Component ID.

Because this is a sensitive operation, the AP requires a Replacement Token to be sent by the host to validate the replacement. The Replacement Token is a 16-byte string that is assigned during the AP's build process. The Replacement Token is used to ensure that the host is authorized to replace the Component.

In order to protect the Replacement Token from being compromised, the AP stores the Replacement Token as three uniquely salted SHA3-512 hashes. The salt is a random 16-byte string that is generated during the AP's build process and is prepended to the Replacement

Token before hashing. Each salt and hash pair is stored in the AP so that the Replacement Token is never stored in plaintext.

Additionally, a fixed transaction delay of 4.5 seconds is sent in place to prevent brute force attacks.

3.4 Boot Verification

The boot verification process is used to ensure that the Application Processor (AP) only boots if all expected Components are present and valid. The AP will verify that each Component is attached and responsive before booting. The AP will then collect each Component's boot message and send it to the host.

3.4.1 BOOT_PING

Sent from the Application Processor to each Component to "ping" the Component to ensure it is attached and responsive. Expected to receive a BOOT_PONG in response.

Name	Offset	Size (bytes)	Content
Magic	`0x00`	80	`\x80` * 80

3.4.2 BOOT_PONG

Sent from the Component to the Application Processor in response to a BOOT_PING. This indicates that the Component is attached and responsive.

Name	Offset	Size (bytes)	Content
Magic	`0x00`	80	`\x81` * 80

3.4.3 BOOT_NOW

Sent from the Application Processor to each Component to command the Component to boot. The Component will then send its boot message (of length 64) to the Application Processor.

Name	Offset	Size (bytes)	Content
Magic	`0x00`	80	`\x82` * 80

[!WARNING]
The component should not respond to a BOOT_NOW if it has not received a BOOT_PING beforehand.

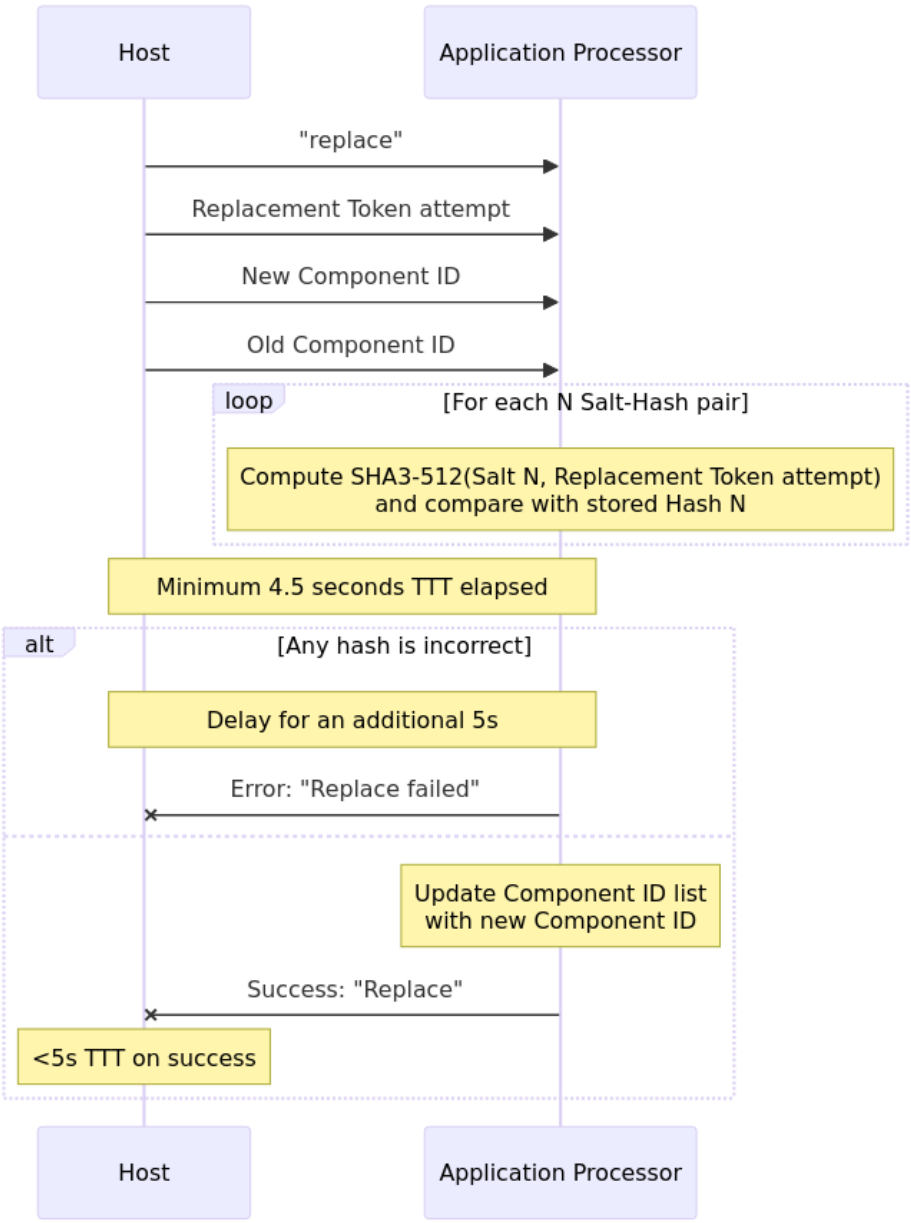

Figure 5: Replace components protocol

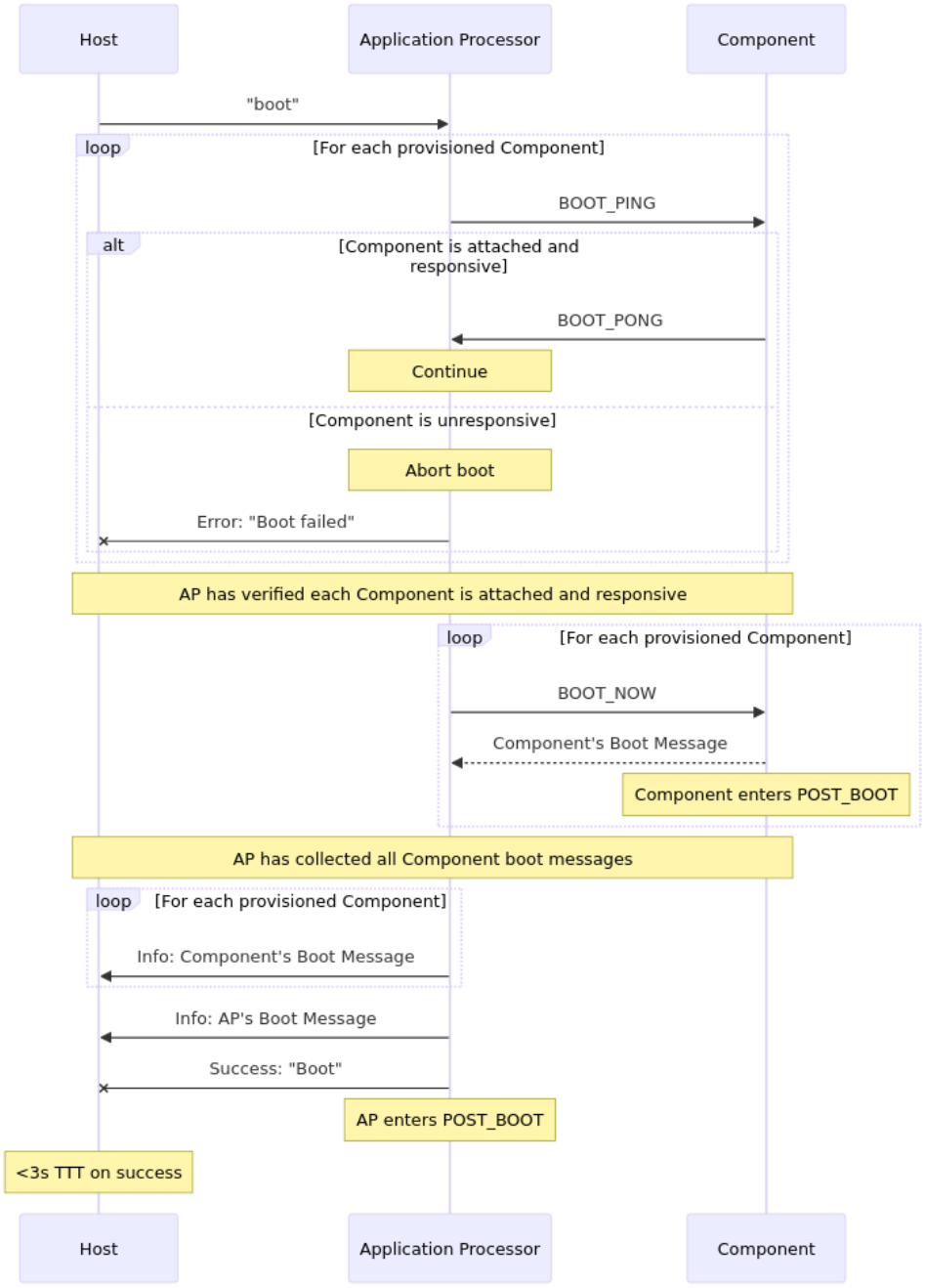

Figure 6: Boot verification protocol

3.5 Post-Boot Communication

As part of the MISC requirements, both the AP and Component must be able to enter a POST_BOOT state, where arbitrary C code runs to support medical device operation. The AP and Component must be able to communicate with each other in this state, so interfaces for HIDE secure send and receive are defined.

For more information about the HIDE protocol, please see the HIDE protocol documentation.

4 Security Requirements

Below we summarize how we achieve the outlined security requirements with our design.

4.1 Security Requirement 1

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

Because of the integrity, authenticity, and non-replayability properties of the HIDE protocol, we ensure that the AP is actively communicating with valid, provisioned Components from the same deployment. A counterfeit Component will not be built on the same deployment and thus will not have access to the Ascon-128 keys required for the encryption layer.

4.2 Security Requirement 2

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

After the AP verifies that each Component is connected and valid, the AP will command each Component to boot. Each Component takes advantage of the integrity, authenticity, and non-replayability properties of the HIDE protocol to confirm that the AP has truly commanded the Component to boot.

4.3 Security Requirement 3

> *"The Attestation PIN and Replacement Token should be kept confidential."*

We use multiple SHA3-512 hashes to validate the Attestation PIN and Replacement Token, preventing compromise of the real PIN and token through side-channel analysis. We also use timers and random delays to effectively prevent brute force and further side-channel attacks.

4.4 Security Requirement 4

> *"Component Attestation Data should be kept confidential."*

Attestation Data is only provided to the AP if the Component receives a command from the AP instructing it to provide Attestation Data. This command is only sent if the AP successfully validates the Attestation PIN. This command cannot be forged due to the properties of the HIDE protocol.

4.5 Security Requiremenet 5

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

The HIDE communication layer uses authenticated encryption with Ascon-128 to verify the authenticity of the sender and receiver. The challenge-response nature of the HIDE protocol and the use of nonces prevents the replay of messages.