2024 MITRE eCTF Design Document

Last Updated: Feb 28, 2024

School	Carnegie Mellon University
Team	Plaid Parliament of Pwning
Team Members	Akash Arun Andrew Chong Aditya Desai Nandan Desai Quinn Henry Sirui (Ray) Huang Tongzhou (Thomas) Liao David Rudo John Samuels Anish Singhani Carson Swoveland Rohan Viswanathan Gabriel Zaragoza
Team Advisors	Anthony Rowe Patrick Tague Maverick Woo

Table of Contents

1 Documentation Notes

This document informs the reader of the functional and cryptographic design implemented by the Plaid Parliament of Pwning in order to secure the Medical Infrastructure Supply Chain (MISC) system in accordance with the functional and security requirements specified in the competition documentation.

Throughout this document, we aim to use a consistent and clear notation. Objects displayed in our protocol diagrams should be interpreted as bit-strings. The double pipe symbol, ||, represents the concatenation of two bit-strings. The braces, { }, represent encryption using the key located in subscript on the outer right side. For example:

$$\{A \,||\, B\}_{Kx}$$

represents the bitstring A concatenated to bitstring B, then encrypted with K_x. We begin by introducing the cryptographic symbols and abbreviations used.

1.1 Principals

AP - Application Processor
C_{id} - Component with ID id
H - Host
AD_{id} - Attestation Data for component C_{id}

1.2 Keys

K_{apbr} - Application Processor boot root key
K_{apbs_id} - Application Processor boot subkey from C_{id}
K_{cbr} - Component boot root key
K_{cbs_id} - Component C_{id} boot subkey
K_{attr} - Attestation root key
K_{atts_id} - Attestation subkey for component C_{id}
K_{apcode} - Application Processor code key
K_{ccode} - Component code key
K_{ssr} - Secure send root key
K_{sss_id} - Secure send subkey for component C_{id}

1.3 Other Symbols

RT - Replacement Token
PIN - Attestation Pin
KDF - Key Derivation Function
POST_BOOT_CODE - Code run after a successful boot
AE - Authenticated Encryption
BB_{ap} - AP Boot metadata blob
BB_{c_id} - Component C_{id} Boot metadata blob
ID - ID of component C_{id}

2 Build Steps

2.1 Build Environment

The build environment includes all dependencies required for a successful build. Dependencies used by the firmware are provided directly in the repo (rather than downloaded during build), to prevent unexpected changes. Dependencies used by the build tools are downloaded but pinned to specific versions, to avoid problems caused by different installation processes between systems. The build itself will only run once and will enable the remaining build stages. We will verify that the tools packages within this environment have no known vulnerabilities.

2.2 Build Deployment

The build deployment stage represents the preparation for fabrication of components to be sent to a specific medical client. This stage will generate a number of "factory" global secrets via the build deployment host tools. These secrets make up a single deployment, and will be passed to the AP and components in subsequent stages to enable security requirements.

The deployment for a given system can be performed via the `ectf_build_depl` host tool.

Global secrets will include:

- The AP boot root key: K_{apbr}
- The Component boot root key: K_{cbr}
- The Attestation root key: K_{attr}
- The Secure Send root key: K_{ssr}
- The Deployment key: K_d

2.3 Build Application Processor

After the environment and deployment are constructed, the AP can be built using the `ectf_build_ap` host tool.

This step will build the AP with the deployment generated in the previous step. At this stage, two secrets are generated:

- The AP boot subkey for each component C_{id} such that $K_{apbs\_id} = KDF(K_{apbr} \,||\, ID)$
- A randomly generated ephemeral AP code key K_{apcode}

Once these secrets are generated, the AP is built with the following:

- The encrypted AP boot root key: $\{K_{apbr}\}_{KDF(RT)}$
- ID's of the Components that its provisioned for which are stored in ascending order.
- The boot metadata blob is encrypted with all K_{apbs_id} subkeys sorted in ascending ID order.
    - The boot metadata blob BB_{ap} contains the boot message, K_{cbr}, $K_{ssr,}$ and K_{apcode}
    - For example, in the case of two components with IDs 1 and 2, the result would be: $\{\{BB_{ap}\}_{K\_apbs\_1}\}_{K\_apbs\_2}$
- The POST_BOOT_CODE encrypted with K_{apcode}: $\{POST\_BOOT\_CODE\}_{K\_apcode}$

- The attestation root key encrypted with KDF(PIN): $\{K_{attr}\}_{KDF(PIN)}$
- The deployment key: K_d
- 8192 bytes (one flash page) of seed entropy for the Random Number Generator

2.4 Build Component

Building the component is nearly identical to building the AP, however this is performed via the `ectf_build_comp` host tool.

During the build component phase, two secrets are generated temporarily:

- The component boot subkey for the given component ID: $K_{cbs\_id} = KDF(K_{cbr} \,||\, ID)$
- The component attestation subkey for the given component ID: $K_{atts\_id} = KDF(K_{attr} \,||\, ID)$
- A randomly generated ephemeral component component code key: K_{ccode}

Once these secrets are generated, the component is built with the following:

- The component attestation data is encrypted like so: $\{AD_{id}\}_{K\_atts\_id}$
- The component boot metadata blob BB_{c_id} encrypted with the component boot subkey: $\{BB_{c\_id}\}_K \mid K = K_{cbs\_id}$
    - The boot metadata contains the component boot message and K_{ccode} for the given component, as well as the component secure send subkey $K_{sss\_id} = KDF(K_{ssr} \,||\, ID)$
- The POST_BOOT_CODE encrypted with K_{ccode}: $\{POST\_BOOT\_CODE\}_{Kccode}$
- Plaintext K_{abps_id}, where $K_{abps\_id} = KDF(K_{apbr} \,||\, ID)$
- The deployment key: K_d
- 8192 bytes (one flash page) of seed entropy for the Random Number Generator

3 System Design

3.1 List Components

The list components functionality lists the IDs of the components currently provisioned and connected to the medical device.

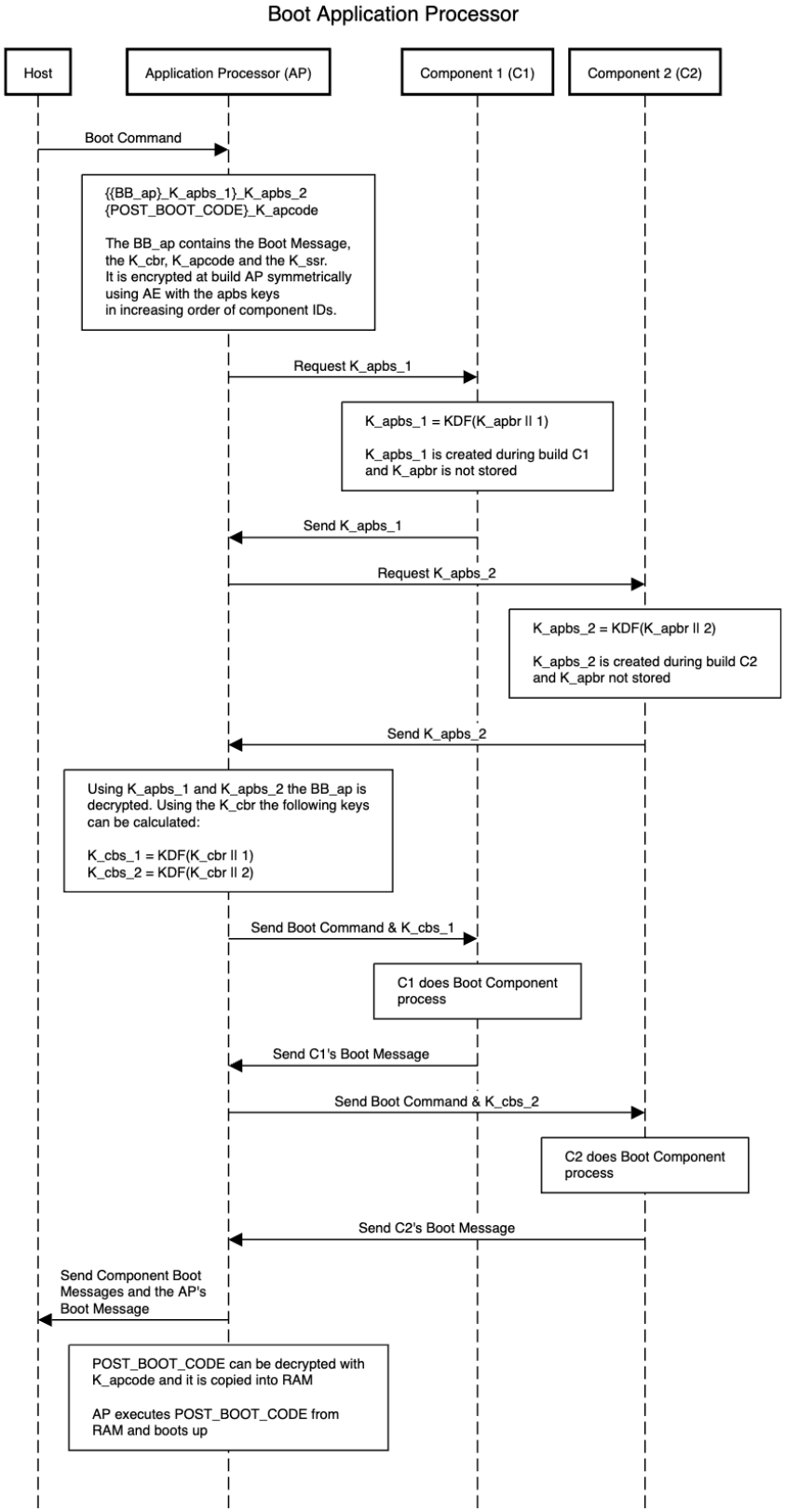
In other words, the AP will print a list of components that are provisioned, and for each possible bus address:

- The AP will send a 0-byte “poll” message to the address. This poll bypasses the Secure Link Layer and is sent in cleartext for performance reasons.
- If the poll is acknowledged, the AP will send a “ping” to the address.
- When a component receives a “ping”, it will respond with a “pong” message, which will include the component’s ID.
- The AP will receive the “pong” message and print out the ID contained within.

3.1.a Security Requirements

There are no security requirements for this feature, since the command may be initiated by anyone and the component IDs are not considered sensitive data.

3.2 Boot Application Processor



Booting the AP works as follows:

- The Host sends a Boot Command to the AP
- AP requests each Component (C_{id}) for its Application Processor ID boot subkey (K_{apbs_id}).
- Each component replies with its boot subkey.
- Once all keys are received from the expected components, the AP uses them to decrypt the AP Boot Metadata Blob (BB_{ap}).
    - The blob is decrypted sequentially with AE in reverse Component ID order.
- If decryption succeeds, the AP begins the Boot Process for the Components.
- For each Component, the AP first sends its component boot key (K_{cbs_id}), waits for the component to reply with its boot message and then forwards it to the host. The component boot keys are calculated with the following formula:
    - $K_{cbs\_id} = KDF(K_{cbr} \,||\, ID)$, where K_{cbr} is extracted from the Boot Metadata Blob.
- Afterwards, the AP releases its own boot message to the host.
- AP then copies the $K_{ssr,}$ into a global variable so its accessible to the POST_BOOT_CODE.
- Using the Application Processor code key (K_{apcode}), which is also extracted from the AP's Boot Metadata Blob, the AP decrypts its own POST_BOOT_CODE
- The AP copies the plaintext code into RAM and executes the POST_BOOT_CODE

3.2.a Security Requirements

SR1: The Application Processor boot ensures that it should only boot if all of the present components are present and valid by first certifying that all of the components that it is provisioned for exist. Then, the Application Processor is able to determine if the Components are valid by using all of the keys returned by the Components to decrypt the boot message and master key. If the decryption fails, this means that there is at least one Component that is not valid and the Application Processor will not boot.

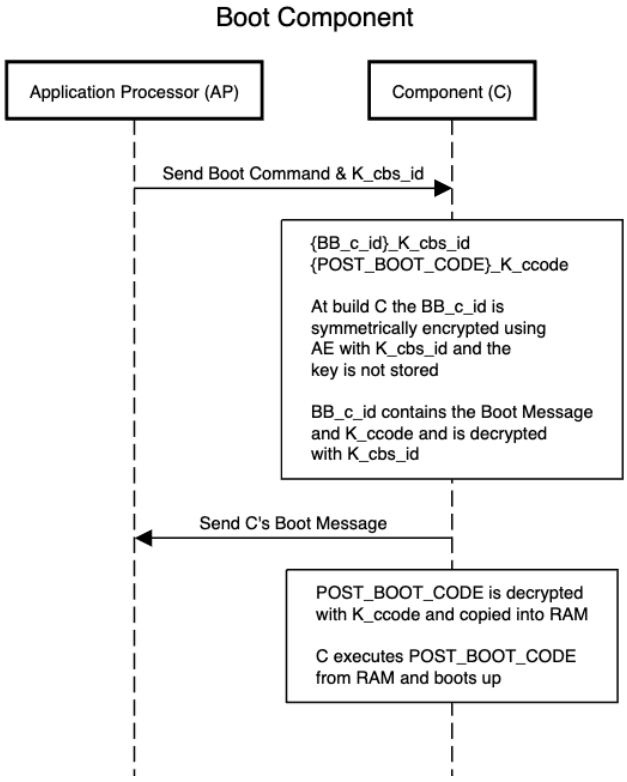
SR2: Does not apply. The Application Processor boot functionality does not refer to how the Components will boot. This will be described more in detail in section 3.7 Boot Component.

SR3: Does not apply. The Application Processor boot functionality does not influence the confidentiality of the Attestation Pin or the Replacement Token.

SR4: Does not apply. The Application Processor boot functionality does not influence the confidentiality of the Component Attestation Data.

SR5: Does not apply. The Application Processor boot functionality does not influence the integrity and authenticity of messages sent and received using the post-boot MISC.

3.3 Boot Component



The boot component functionality works as follows:

- The component receives a boot command and its Component boot subkey (K_{cbs_id}) from the AP, which is used to decrypt its own Boot Metadata Blob (BB_{c_id}) using AE.
- If decryption succeeds, it extracts the boot message from the decrypted Boot Metadata Blob and replies back to the AP with the boot message.
- The component copies K_{sss_id} from BB_{c_id} and stores it in a global variable so that the POST_BOOT_CODE can access it.
- The Component extracts K_{ccode} (unique to each component) from the metadata blob to decrypt its own POST_BOOT_CODE using AE.
- If decryption succeeds, the Component Copies the plaintext code into RAM and executes the POST_BOOT_CODE.

3.3.a Security Requirements

SR1: This does not apply, since it happens after the Application Processor boots.

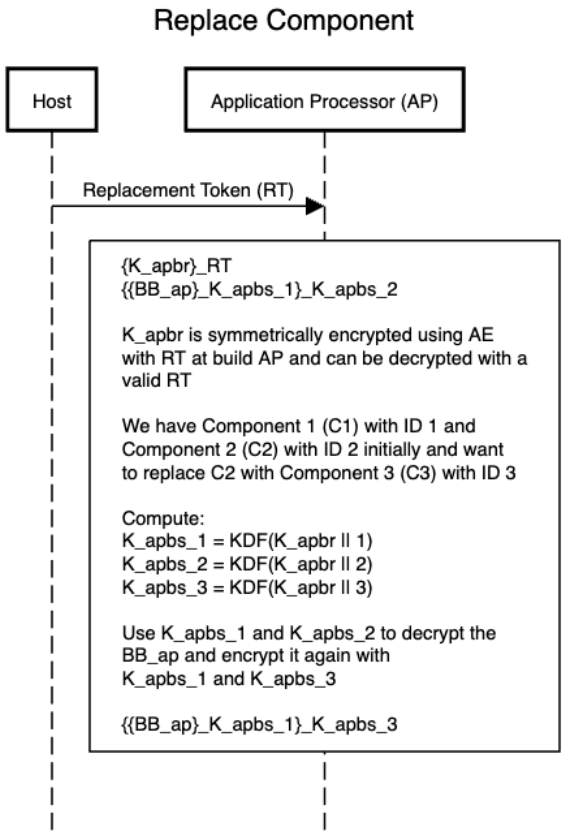
SR2: Component boot does ensure that it only boots after being commanded by a valid Application Processor since only a valid Application Processor can compute the correct component boot key from K_{cbr}. This also only happens after the Application Processor boots so the integrity of the Component has already been checked.

SR3: This does not apply, since Attestation PIN and Replacement Token aren't used to boot the component.

SR4: This does not apply, since Attestation Data isn't sent during the Component boot process.

SR5: This does not apply, since it happens before the component boots so before the secure communication functionality is needed to be activated.

3.4 Replace Component



Note: In the following description, assume the Host is attempting to keep C_1 and replace C_2 with a new component C_3

The replace component functionality works as follows:

- Upon receiving the current Replacement Token from the Host, the AP will attempt to decrypt $\{K_{apbr}\}_{RT}$ with the given Replacement Token using AE.
- If decryption succeeds, the AP computes a key for each component using the following formula:
  - $K_{apbs\_id} = Hash(K_{apbr} \,||\, ID)$
- Using the AP's provisioned component keys, it attempts to decrypt $\{\{BB_{ap}\}_{K_{apbs\_1}}\}_{K_{apbs\_2}}$ by applying keys sequentially from reverse component ID order.
- If the boot data is successfully decrypted for both the Boot Data and Master Key, the Boot Data is encrypted again sequentially with the new component keys in component ID using the following formula:
  - $\{\{BB_{ap}\}_{K_{apbs\_1}}\}_{K_{apbs\_3}}$

3.4.a Security Requirements

SR1: The Replace Component functionality does influence the booting functionality of the Application Processor. If the Replace Component functionality were compromised and allowed the application processor to provision for an improper component, the Application Processor will be able to boot without the proper component. By protecting the confidentiality of the Replacement Token by using it as an encryption key rather than storing the value in memory, this threat is ameliorated. Note that for our system, even if the confidentiality of the Replacement Token were compromised, the malicious Component would only be able to boot the AP by replying to the AP with the correct subkey (calculated using the Application Processor boot root key).

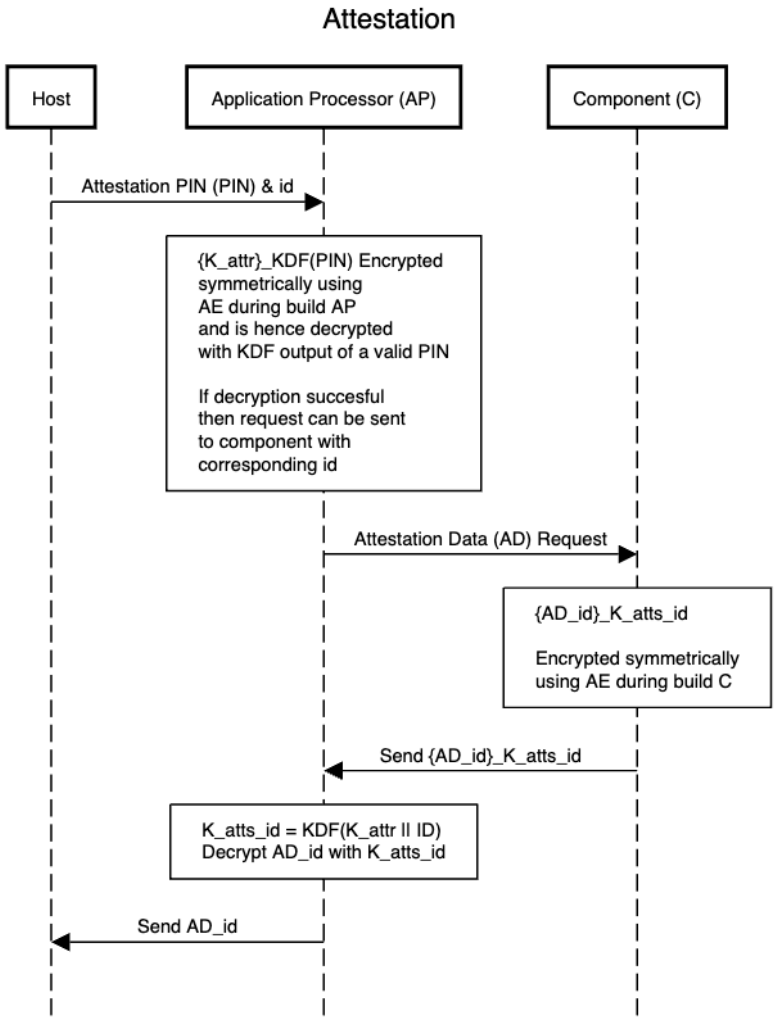
SR2: Does not apply. The Replace Component functionality does not refer to how the Components will boot.

SR3: The Replace Component functionality does not influence the confidentiality of the Attestation Pin. However, this functionality is completely dependent on the confidentiality of the Replacement Token. The Replacement Token itself is not stored on the device, but is used as an encryption key to encrypt the Master Key. If the incorrect Replacement Token is given in the command, it is not possible to recover the Replacement Token given the current functionality.

SR4: Does not apply. The Replace Component functionality does not influence the confidentiality of the Component Attestation Data.

SR5: Does not apply. The Replace Component functionality does not influence the integrity and authenticity of messages sent and received using the post-boot MISC.

3.5 Attestation



The attestation functionality works as follows:

- Host sends Attestation Pin (PIN) and Component ID (C_{id}) to the AP
- If it is verified that C_{id} is provisioned for by the AP, AP attempts to decrypt the Attestation Root Key ($\{K_{attr}\}_{KDF(PIN)}$) using the given Attestation Pin using AE.
- If decryption succeeds, the AP sends an Attestation Data request to the Component.
- Once the component receives the request, it replies with the Attestation Data encrypted with the Attestation Subkey ($\{AD_{id}\}_{K\_atts\_id}$) back to the AP.
- The AP derives the Attestation Subkey as $K_{atts\_id} = KDF(K_{attr} \,||\, ID)$.
- The AP decrypts the attestation data from each component using AE.
- If the decryption of each component succeeds, the Application Processor will forward it to the Host.

3.5.a Security Requirements

SR1: Does not apply. The Attestation functionality does not influence the booting functionality of the Application Processor.

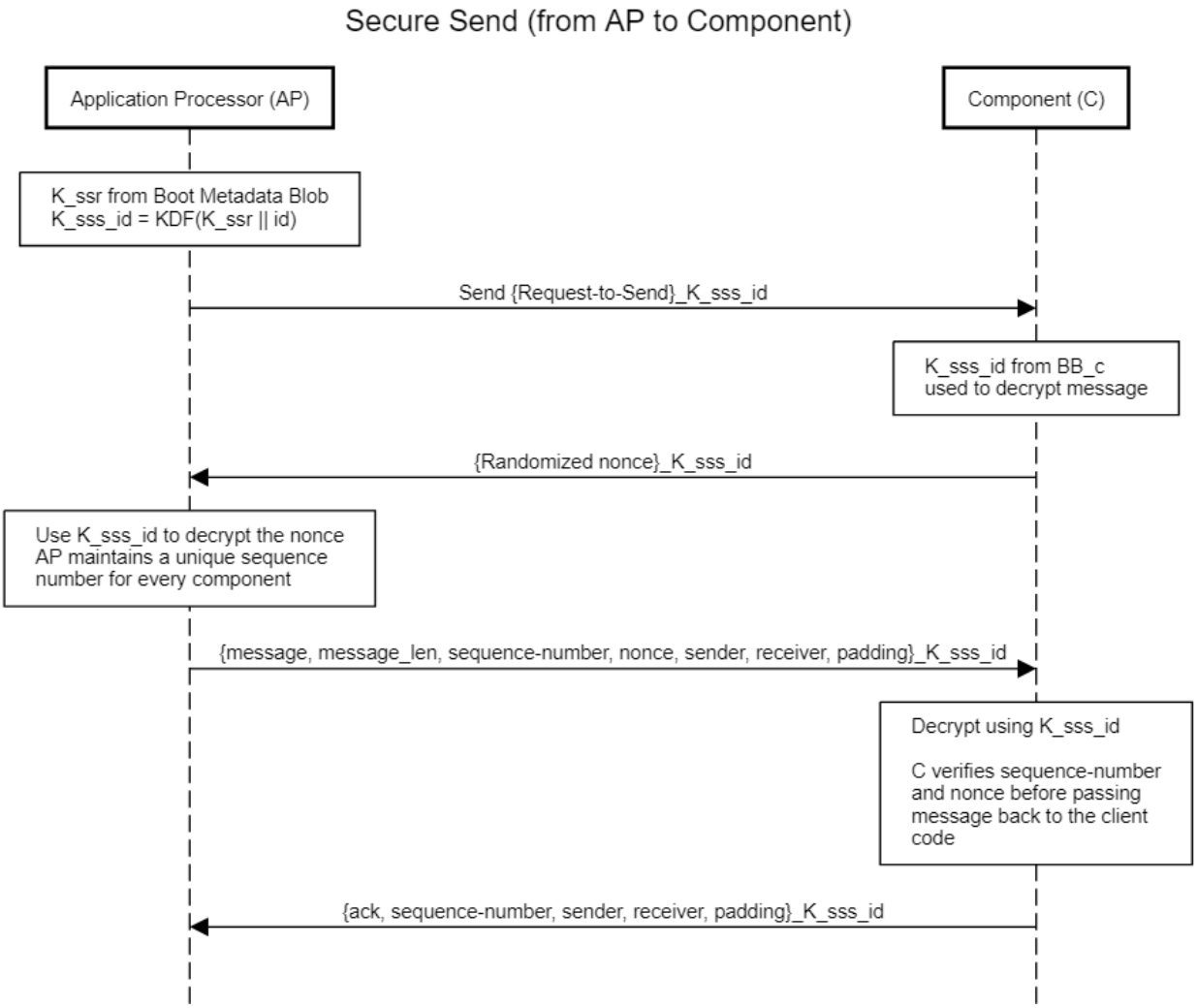
SR2: Does not apply. The Attestation functionality does not refer to how the Components will boot.

SR3: The confidentiality of the Attestation Pin must be protected to ensure proper Attestation. However, the Attestation Pin is only used to encrypt the Attestation Key, but not stored in memory.

SR4: The confidentiality of the Component Attestation Data is protected using multiple methods. The Attestation Data on each Component is encrypted by an Attestation Key, which is only stored on the Application Processor, which means it is not possible to retrieve the Component Attestation Data from just the Component itself. Furthermore, the Attestation Key is also only ever stored encrypted using AE. It can only be decrypted by the correct Attestation PIN. Otherwise the Attestation Functionality will fail.

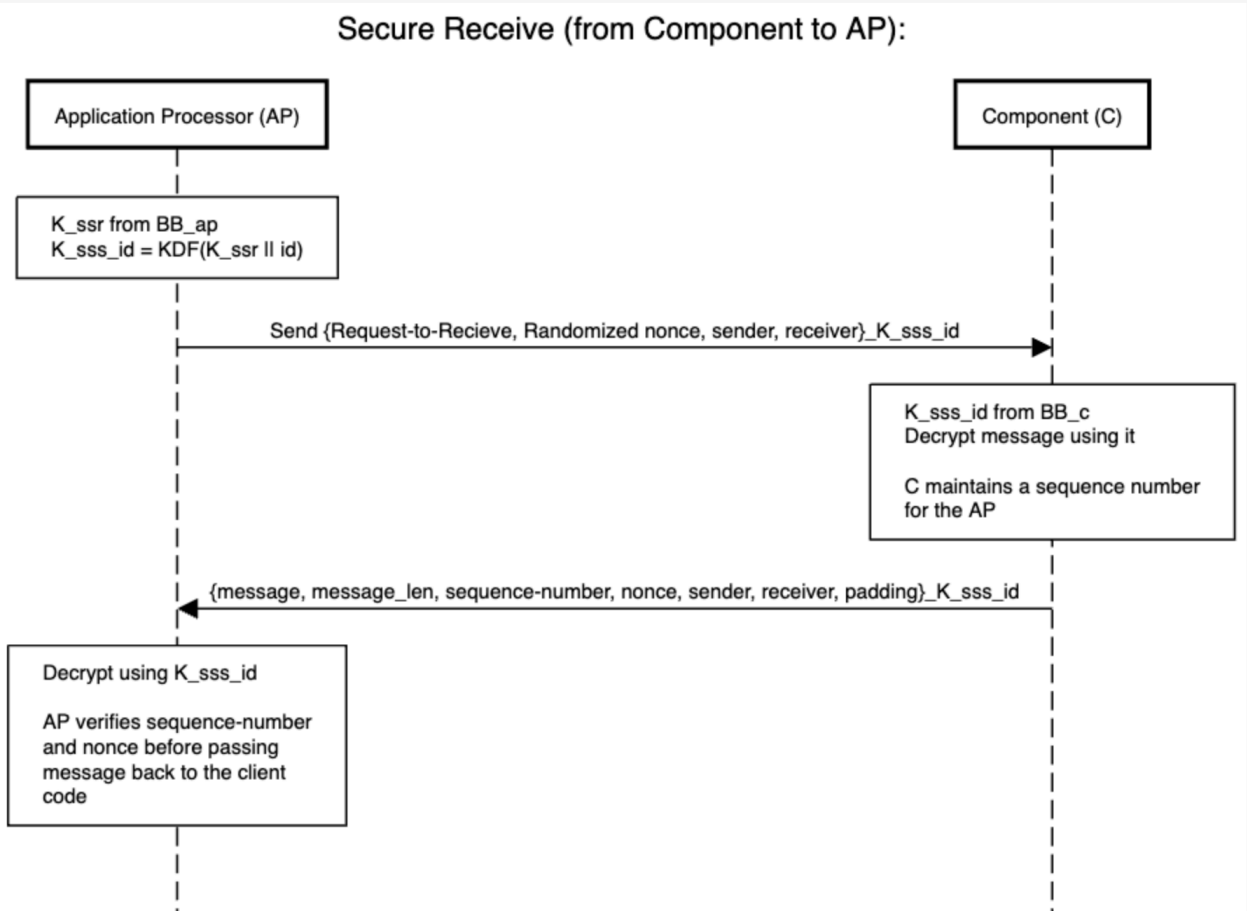
SR5: Does not apply. The Attestation functionality does not influence the integrity and authenticity of messages sent and received using the post-boot MISC.

3.6 Secure Send & Receive



Secure Send (From AP to Component):

- AP sends a Request-to-Send message to Component C_{id}
- Component C_{id} replies with a randomized nonce, encrypted using the K_{sss_id} subkey
    - *Note: K_{sss_id} is derived by the AP from root key using the following formula:*
        - $K_{sss\_id} = KDF(K_{ssr} \,||\, ID)$)
- AP responds with a message formatted as:
    - {message, sequence-number, message length, nonce, sender, receiver, padding}.
    - The message is encrypted using K_{sss_id}
    - *Note: Sequence number on the AP is maintained per component*
- Component verifies correctness of sequence-number and nonce, before responding to the AP with an acknowledgment message
    - The received message, after verification, is then passed back to the client code.



Secure Receive (from Component to AP):

- AP sends a Request-to-Receive message including a randomized nonce and component's ID, encrypted using the K_{sss_id} subkey
- Component responds with a message with the following format:
    - {message, sequence-number, nonce, sender, receiver, padding}
    - The message is encrypted using K_{sss_id}
- AP verifies correctness of sequence-number and nonce, before passing the message back to the client code.

Best-Effort Retry Protocol:

The secure send/receive protocol makes a 'best effort' attempt at retrying transmission/reception when errors occur in either sending/receiving packets or in the processing of each packet in respect to the above protocol expectations. If the message gets corrupted during the transit and fails to decrypt as a result, the AP/Component will try to resend the message. The AP/Component will also try to resend the packet in case of sequence number mismatch or nonce mismatch. The recipient of the message will also go back to listening state, in case of a failed message.

3.6.a Security Requirements

SR1: Does not apply. The secure send and receive functionality does not influence the booting functionality of the Application Processor.

SR2: Does not apply. The secure send and receive functionality does not refer to how the Components will boot.

SR3: Does not apply. The secure send and receive functionality does not relate to the confidentiality of the attestation Pin.

SR4: Does not apply. The secure send and receive functionality does not relate to the confidentiality of the attestation data.

SR5: The integrity and authenticity of post-boot communication is ensured using a handshake to establish a nonce paired with symmetrically encrypted messages between the principals. This ensures messages cannot be manipulated or replayed. Additionally, a sequence-counter is used to prevent messages from being dropped, duplicated, or reordered.

4 Security Design

4.1 Cryptography

Our system uses symmetric authenticated encryption in several places, to protect confidential data. Most confidential data is stored encrypted at rest, including internal keys, boot messages, and post-boot code. We use the XChaCha20-Poly1305 scheme, with a small modification to make it nonce-misuse-resistant. Specifically, we generate our nonces by taking a hash of the plaintext message, and combining this with a random nonce from our RNG. This allows our encryption to be resistant against attacks on the RNG (even if the RNG is compromised, we will never use the same nonce twice with different messages).

We are using BLAKE2b as our Key Derivation Function (KDF). Component-specific subkeys are derived by hashing a concatenation of the root key and the component ID. KDFs for the PIN and Replacement Token are computed using a large number of hash iterations on the plaintext. To resist brute-force attacks, we use the maximum number of iterations that can fit within the time requirements.

4.2 Attestation PIN and Replacement Token Confidentiality

The Attestation PIN and Replacement Token are utilized to derive symmetric keys via a BLAKE2b-based KDF. Neither the plaintext PIN or Replacement Token, nor any hashes thereof, are ever loaded onto any device. This improves our resistance against memory-based and side-channel attacks.

As discussed in the Attestation section, the KDF of the PIN is used to decrypt the attestation root key, which is then used to decrypt the attestation data itself. Since we use authenticated encryption, failing this decryption process is sufficient to indicate that the PIN is incorrect without needing to explicitly check the PIN.

A very similar process is used for the Replacement Token, which is used to decrypt the AP boot root key. Similarly to the attestation key, failing to decrypt the boot root key indicates that the token is incorrect.

4.3 TRNG Design

For secure random number generation, we use the MAX78000's built-in True Random Number generator as a source of true entropy. To reduce the risk of an attacker disabling the TRNG, we also incorporate the current system tick counter and a previously-stored entropy pool from the previous generator. To add additional confidence to our random number generator, a debiasing algorithm is added on top of the TRNG to ensure a uniform distribution of bits are generated.

Our RNG provides two different versions of generation based on security needs. There is a "secure" version, which extracts a high amount of entropy and uses flash to ensure the entropy pool is stored across iterations. Additionally, there is a "fast" version, which extracts additional randomness from a

smaller amount of entropy and stores its pool in RAM. This is necessary for protocols like messaging, which require low latency to function properly.

The “secure” random number generation works as follows:

- Get data from TRNG
    - Debias and hash it (BLAKE2b) for further computation
- Get current system ticks
    - Hash it using BLAKE2b
- Get previously computed value from flash memory
- Compute the 3 values above hashed together
    - This is stored as the next previously computer value in flash memory
- Hash one more time and return the result

The “fast” random number generator works as follows:

- If we’ve exceeded a certain number of generations, regenerate the RAM entropy pool
    - Uses the “secure” RNG to update this pool
- Get unfiltered TRNG data, current system ticks, a counter for the number of uses after regeneration, and the current RAM entropy pool
- Hash (BLAKE2b) the current pool, ticks, and counter appended together => new RAM entropy pool
- Hash (BLAKE2b) the current pool, ticks, counter, TRNG data appended together => result

4.4 Link Layer

We design and implement StruggleBus, a bus protocol that acts as a data link between the AP and Components, with the goal of supporting functionality and reducing attack surface. The protocol operates on two pulled-up wires, making it compatible with the circuitry used by I^2C. To mitigate hardware-based attacks, we provide a pure-software implementation of StruggleBus that is free from the use of specialized hardware peripherals apart from timers and GPIO. In addition, interrupts are not used in our implementation.

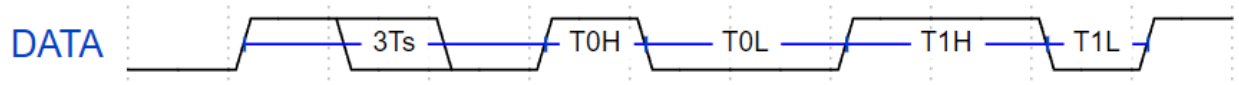
4.4.a Bus Topology

The StruggleBus protocol supports communication between bus participants, including a single AP and any amount of Components, subject to the limit of the address space (see 4.4.e). All participants share two pulled-up wires and a common ground. Specifically, any valid standard-mode I^2C-compatible physical connection without I^2C multiplexers and switches is a valid StruggleBus connection. The two shared wires are designated as DATA and SYNC, which is used for data transfer and acknowledgement, respectively.

Like I^2C, only the AP can initiate a round of communication. In each round, the AP sends one packet (see 4.4.d) to a Component. The component then either responds with one packet, or does not respond at all. Whether or not the component responds can be determined by the content of the packet, which [†]StruggleBus does not care about. However, the AP and the Component must agree on the condition in which the Component should respond, and such conditions must be deterministic with respect to the contents of the packet.

4.4.b Data Signals

The DATA wire uses pulse-width modulation (PWM), which represents bits as the width of pulses. The width of the pulse is derived from the sample time T_s that is agreed upon by all participants. A '0' bit is represented by a logical HIGH of length T_{0H} followed by a logical LOW of length T_{0L}. Similarly, a '1' bit is represented by a logical HIGH of length T_{1H} followed by a logical LOW of length T_{1L}. A stop condition is a logical HIGH of length at least T_{stop}. Timing parameters can be chosen by the implementation, but must satisfy $T_{0H} + T_{0L} = T_{1H} + T_{1L} = kT_s$ where $k \geq 3$ is an integer.



In our implementation, we choose the following timing parameters to achieve a nominal bitrate of 100kbps, identical to that of standard-mode I^2C.

Parameter	Value
T_s	3.33µs
T_{0H}	T_s
T_{0L}	$2T_s$
T_{1H}	$2T_s$
T_{1L}	T_s
T_{stop}	$3T_s$

When the AP does not need to send data and does not expect to receive data, it drives the DATA wire LOW, making Component-to-Component communication, which is non-compliant, impossible during this period. This acts as a defense-in-depth measure.

StruggleBus uses PWM to represent data mainly due to the need to detect timing violations on the bus to prevent intentional modification of the data. Timing violations can rarely happen naturally on a single-threaded real-time embedded system with interrupts disabled. Therefore, detecting timing violations is an adequate defense-in-depth measure against malicious devices on the bus. Any synchronous two-wire protocol would require at least four time measurements per bit to enforce timing on both wires, because there is a rising edge and a falling edge to measure on both wires. Any one-wire protocol that requires one sample per bit, like UART, requires explicit clock synchronization between the sender and the receiver, which adds complexity and decreases robustness. One-wire protocols that require two samples per bit typically utilize Manchester encoding, which requires recovering the clock from the signal before decoding it, again adding complexity. PWM, as utilized by StruggleBus, has the unique advantage of being able to decode valid data starting from any rising edge by measuring the time until the next falling edge, independent to the rest of the signal, and making timing violation detection trivial. It requires at least three samples per bit, which is less than that of a two-wire synchronous protocol if timing enforcement is required. In addition, this frees up a wire for acknowledgment, which separates data and control physically, making the content on each wire unambiguous. Therefore, we determine PWM to be the optimal bit representation for our use case.

4.4.c Acknowledgment Signals

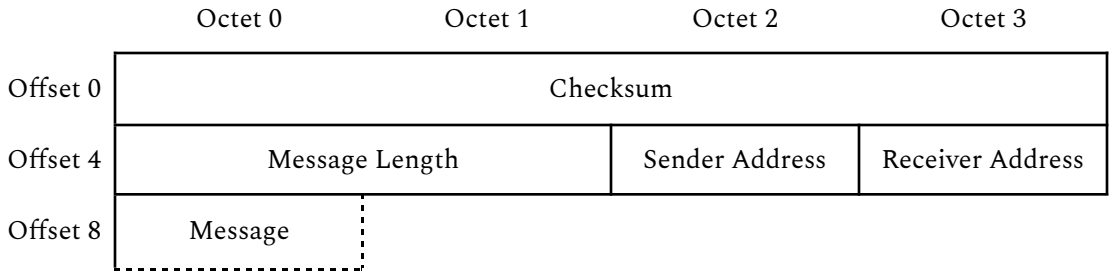
Acknowledgement in StruggleBus is analogous to that of I^2C. However, only the Component acknowledges packets, and only the AP expects acknowledgements. This is because the other direction is never necessary for our use case.

Like the DATA wire, the SYNC wire is normally driven LOW by the AP to prevent non-compliant Component-to-Component communication. When the AP-sender finishes sending a packet, it releases

the SYNC wire, which will be pulled HIGH by the pull-up resistor. When the Component-receiver receives a valid packet, it drives the SYNC wire LOW for T_{ack}, and then releases it. The AP-sender detects a LOW on the SYNC wire as a valid acknowledgement and drives the SYNC wire LOW, restoring its initial condition. In our implementation, we choose $T_{ack} = 3T_s$.

4.4.d Packet Format

A StruggleBus packet is little-endian and has an integer number of octets. A packet always starts with a fixed-size header, followed by a variable-size message.



The header consists of a 32-bit checksum (see 4.4.f), a 16-bit message length, a 8-bit sender address, and a 8-bit receiver address (see 4.4.e). The message is a byte stream that is at most 2^{16} - 1 octets. In our implementation, we limit the maximum size of the message to 400 octets, because larger messages are unnecessary for our use case.

4.4.e Addressing

Each participant on the StruggleBus must have a unique and valid address. The AP's address is always 0. A Component's address must be a valid I^2C address. When a participant receives a packet with a receiver address that does not match their own, it must ignore that packet but not raise an error.

4.4.f Checksumming

The integrity of a StruggleBus packet can be verified with a 32-bit checksum, which we choose to be the FNV-1a[1] non-cryptographic hash function. The checksum is calculated from Octet 4 of the header (the first octet of the message length) to the last octet of the message (both bounds are inclusive).

FNV-1a is chosen for its extreme simplicity and relatively good effectiveness as a checksum. StruggleBus intends to use the checksum to detect natural bit-flips, but not intentional tampering.

4.5 Secure Link Layer

As a defense-in-depth measure, we implement a Secure Link Layer protocol called Provably Paranoid Protocol (PPP), which encrypts and authenticates all communications between the AP and Components. The authentication property of this allows each device to ensure that it is only

[1] http://www.isthe.com/chongo/tech/comp/fnv/

communicating with other valid devices from the same deployment. It also resists black-box brute-force attacks against the Component, and generally prevents the attacker from tampering with internal communications. The confidentiality property provides additional resistance against protocol analysis on a live system.

The Secure Link Layer is implemented as a transparent wrapper around the Link Layer. Before sending each message, it adds a metadata header (source, destination, length, and a nonce), and then encrypts the message using our authenticated encryption scheme. On reception, the message is decrypted and validated before returning to the client code. Because messages are always sent in pairs (one message from the AP gets one response from a Component), we are able to generate a nonce in each message from the AP, and echo back the nonce in the message from the Component. This allows the AP to validate that messages from the Component are not being replayed (it does not make any guarantees in the opposite direction, which must be handled by upper layers in the protocol).

5 Defenses

5.1 External Code Removal

To minimize possible attack surfaces, we remove as much as possible of the Maxim SDK and Newlib. This prevents us from accidentally using complex or buggy features that may be present in MSDK and Newlib, and massively reduces the amount of code that we need to audit. Any feature that we do need can be individually audited, or reimplemented in a way that better suits our system. Additionally, we simplify our build process by maintaining a copy of all design dependencies (MSDK, MUSL, Monocypher) in our design repository, to prevent issues related to upstream versions changing.

5.2 Fault Injection Countermeasures

Because an attacker is expected to have physical access to tamper with our hardware, we harden our critical sections against fault-injection attacks. This is done by repeating critical calculations multiple times, along with computing known "padding" computations in between these calculations. These calculations are checked for self-consistency, and if the system detects any self-inconsistency (i.e. adding two known numbers not producing the correct result) then it will assume that there has been a hardware failure; and enter into an error state (a spin state requiring a power-on reset). Sensitive computations such as encryption and memory copies also include an extra verification pass, to ensure that data was not corrupted during the operation.

The only condition that will trigger this error state is if an internal hardware failure is detected. It is not possible to enter this error state through only external inputs (i.e. no sequence of data over GPIO, UART, or USB can cause the system to enter such a state). Small, randomized time delays (generated using our TRNG) are also inserted immediately before important calculations in order to significantly increase the difficulty of corrupting the given calculations via a fault attack. Finally, cryptographic signing operations are immediately verified (after signing) using the same set of keys in order to detect if there was any hardware-level corruption or failure.

5.3 Bruteforce Countermeasures

To prevent brute force attacks, the system has protection measures in place for both the AP and components.

When validating a PIN or replacement token, we use a large number of KDF iterations to resist online brute-force attacks while still remaining within the 3-second timing requirements. For attestation, we also require the AP to communicate with the Component before verifying the PIN, further increasing the cost of a brute-force attack. After we have detected an incorrect PIN or replacement token, we delay for an additional 4 seconds before further attempts. This counter is stored in flash to ensure it persists even if the device is power-cycled. The counter is repeatedly updated during the delay, to ensure that the delay can never be more than 5 seconds.

5.4 Memory and Execution Countermeasures

In the unlikely event of an attacker gaining write access to SRAM or partial control of the instruction pointer, the system has hardware memory protection enabled. The Memory Protection Unit (MPU) restricts execution to only those regions of memory that are expected to contain code, and prevents write access to executable sections of SRAM.