MITRE eCTF 2024 Design Document

Team Sp4rtans
School Michigan State University

Members

Udbhav Saxena
Charles Selipsky
Aashish Harishchandre
Fatima Saad
Krishna Patel
Radhe Patel
Felipe Marques Allevato
Riley Cook
Aditya Chaudhari
Samay Achar
Ramisa Anjum

Advisors

Dr. Qiben Yan
Adrian Self

Last Updated: **2024-03-13**

Contents

1 Syntax and Symbols

In this document, we will try to explain our implementation in a clear and coherent manner. For that purpose, we will define some of the notations used throughout the document.

1.1 Syntax

$h(x)$: Hash function applied on data x.

$\{x\}_K$: Encryption/Decryption of data x using key K.

$x + y$: Concatenation of data x and y.

1.2 Symbols

$AP_{1,2}$: Application Processor built for components 1, and 2.

C_1: Component with an ID represented by a number subscript (e.g. 1 is x24 in the example code).

M: Manufacturer's side, comprising of keys generated before $AP_{1,2}$ and C_1 are built.

DH: Ephemeral Diffie-Hellman key exchange process done in handshake.

$ID_{AP_{1,2}}$: Unique Identifier of that entity (In this case of $AP_{1,2}$).

IV: Initialization Vector for our Authenticated Encryption with Additional Data (AEAD) algorithm (`chacha20_poly1305`).

AAD: Additional Authentication Data for AEAD.

1.3 Cryptographic Keys

SK_M: Manufacturer's Secret Key used for certificate signing and never stored in the boards.

PK_M: Manufacturer's Public Key used for certificate verification.

$SK_{AP_{1,2}}$: $AP_{1,2}$'s Secret Key used to verify challenge sent from C_1 and C_2.

$PK_{AP_{1,2}}$: $AP_{1,2}$'s Public Key that signs challenge sent to C_1 and C_2.

SK_{C_1}: C_1's Secret Key used to verify challenge sent from $AP_{1,2}$.

PK_{C_1}: C_1's Public Key that signs challenge sent to $AP_{1,2}$.

$SK^{DH}_{AP_{1,2}}$: $AP_{1,2}$'s Secret DH Key used for the symmetric key derivation.

$PK^{DH}_{AP_{1,2}}$: $AP_{1,2}$'s Public DH Key used for the symmetric key derivation and challenge response.

$SK^{DH}_{C_1}$: C_1's Secret DH Key used for the symmetric key derivation.

$PK^{DH}_{C_1}$: C_1's Public DH Key used for the symmetric key derivation and challenge response.

$K_{C_1,AP_{1,2}}$: C_1 and $AP_{1,2}$'s shared symmetric key derived from DH and used in communications.

2 Key Elements

2.1 Public Key Cryptography

Public-key cryptography will be used as the central system to ensure that application firmwares and components in a system are genuine. In our setup, each deployment will correspond to a manufacturer keypair stored in the global secrets for that deployment, acting as the root certificate authority. We utilize Ed25519 with a specific key size, leveraging wolfSSL wolfCrypt primitive functions for cryptographic operations.

Every subsequent application and component will get its own keypair generated at build time, where a package composed of the device's public key along with its identifying information will be signed by the manufacturer's public key to certify its authenticity (analogous to an X.509 certificate).

Utilizing functions such as `'Make_curve25519_key'` and `'Construct_device_cert_data'` we were able to construct those key pairs and certificates (More on Section 4.1.1). This leads to the functions like `'Load_ap_private_key'` and `'Load_comp_private_key'` that securely load the private key of the AP and components, limiting access to the secret keys.

In this arrangement, every device only stores its own secret key. This ensures that the compromise of secrets by a single component in the system does not leak information about other parts or make them vulnerable, strengthening the security of the MISC as a whole.

2.2 Entropy

For the purpose of utilizing resilient entropy source for our processes, we make use of the MAX78000's TRNG features imported from the Maxim-supplied Universal Cryptographic Library (UCL).

This promises to generate random data using one or more internal noise sources from the board that is then processed by software and hardware and returned by the UCL function.

The generated data is "intended to meet the requirements of common security validations" as per the manufacturer's user guide, and we believe it will provide sufficiently random entropy compared to custom code-based implementations.

2.3 Communication

Secure board communications are established using our custom implementation of a TLS handshake with mutual authentication of component and application certificates. This utilizes an Ephemeral Diffie-Helmann key exchange to ensure forward secrecy. (More on the handshake in 4.1.1)

Once this is done, a session is estabilished and the communications are done using the symmetric ChaCha20 stream cipher combined with the Poly1305 message authentication mechanism to establish communication using Authenticated Encryption with Additional Data (AEAD) (Section 4.2).

3 Build Processes

3.1 Build Deployment Phase

When a deployment is being built for a particular MISC, the global secrets generated are read-only and cannot be modified later while building application firmwares and components.

Because of this restraint, we chose to generate a single key pair representing the manufacturer's credentials, where the public key will be used to certify authenticity and the private key will not be distributed to any built device. These are the following:

$$SK_M,\ PK_M$$

3.2 Build Application Processor

When building an application processor, the build tool will generate an Ed25519 keypair for signing and verification:

$$SK_{AP_{1,2}},\ PK_{AP_{1,2}}$$

It will use the SK_M from the global secrets to sign a certificate comprising of a concatenation of a prefix unique to the Application Processor along with it's public key:

$$\{h(ID_{AP1,2} + PK_{AP_{1,2}})\}_{SK_M}$$

3.3 Build Component

When building a component from a deployment, the build tool will similarly generate a unique Ed25519 key pair for the component:

$$SK_{C_1},\ PK_{C_1}$$

Then, it will sign a certificate of the concatenation of the Component's ID and its public key with SK_M:

$$\{h(ID_{C_1} + PK_{C_1})\}_{SK_M}$$

4 Functionalities

In this section, we will discuss our implementation's security of the functional requirements. Therefore, we will go over the communication of the boards, their security features, and how the functionalities work in our design.

4.1 Boot

When issuing the boot command, the AP will do our custom TLS handshake with each component connected to it to ensure authenticity and will generate a confidential shared secret for a symmetric communication to be estabilished.

4.1.1 Handshake

In order to explain the handshake, we will consider $AP_{1,2}$ and C_1. The process starts with the $AP_{1,2}$ sending a hello message, which is created with a `signed_hello_with_cert` struct.

Both sides first generate their keys for Ephemeral Diffie-Hellman ($PK^{DH}_{AP_{1,2}}$ and $PK^{DH}_{C_1}$) using `Curve25519`.

This message contains the $PK_{AP_{1,2}}$, $PK^{DH}_{AP_{1,2}}$, and a signature of the two using $SK_{AP_{1,2}}$ to ensure integrity of the message and proving that $AP_{1,2}$ owns the DH key, as follows:

$$PK_{AP_{1,2}} + PK^{DH}_{AP_{1,2}} + \{h(PK_{AP_{1,2}} + PK^{DH}_{AP_{1,2}})\}_{SK_{AP_{1,2}}}$$

The message also includes a certificate created at build time signed by SK_M (Section 3.2), showing that the manufacturer certified $AP_{1,2}$:

$$\{h(ID_{AP1,2} + PK_{AP_{1,2}})\}_{SK_M}$$

Now, C_1 receives the message and validates $PK_{AP_{1,2}}$ from the certificate generated by the manufacturer using the PK_M. This can then be used to verify the hello message sent from $AP_{1,2}$ using the validated $PK_{AP_{1,2}}$.

Going from the other direction, C_1 also sends a hello in the same structure, using the PK_{C_1}, $PK^{DH}_{C_1}$, and a signature of the two using SK_{C_1}:

$$PK_{C_1} + PK^{DH}_{C_1} + \{h(PK_{C_1} + PK^{DH}_{C_1})\}_{SK_{C_1}}$$

Similarly, C_1 includes its' own certificate created in build time (Section 3.3):

$$\{h(ID_{C_1} + PK_{C_1})\}_{SK_M}$$

To avoid replay attacks, we use the Ephemeral Diffie-Hellman (DH) keys as uniquely random nonces in a challenge-response mechanism. So, C_1 encrypts $PK^{DH}_{AP_{1,2}}$ using SK_{C_1} and sends it to $AP_{1,2}$:

$$\{PK^{DH}_{AP_{1,2}}\}_{SK_{C_1}}$$

Finally, $AP_{1,2}$ receives the message, validates the certificate and message signature, and derives the

$K_{C_1,AP_{1,2}}$ in a similar fashion. The $AP_{1,2}$ finishes the handshake by sending its challenge-response:

$$\{PK_{C_1}^{DH}\}_{SK_{AP_{1,2}}}$$

On completion of the handshake, both parties create and initialize a session that stores the shared key.

As an overview of the handshake, here's a diagram of the whole process:

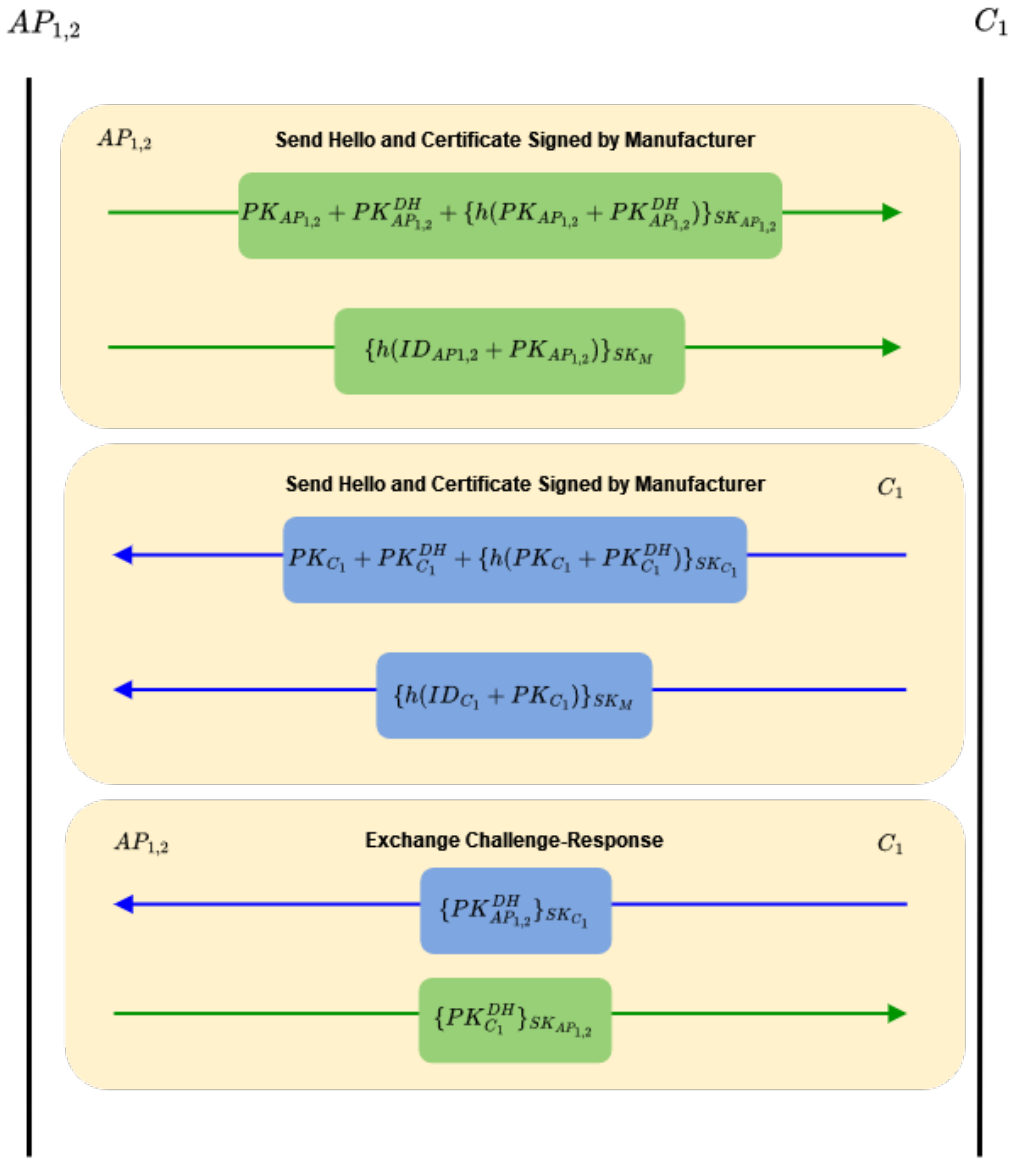

Figure 1: Custom TLS Handshake Diagram

4.2 Post Boot

In the Post Boot part of the design, the functionality is injected by the organizers, and thus we can't secure the injected code itself.

However, every communication uses our implementation of `secure_send` and `secure_receive`.

4.2.1 AP to Component

This subsection will focus on describing the $AP_{1,2} \rightarrow C_1$ direction of the communication.

First, the $AP_{1,2}$ checks if the desired component is connected to the board and has a session active. The session was established in the end of our custom handshake defined in section 4.1.1.

Then, it creates a nonce (Section 2.2) as the 12 byte IV of the `chacha20_poly1305` encryption and the AAD (Authentication Data), utilizing them to encrypt the desired message x:

$$IV, AAD \rightarrow \{x\}_{K_{AP_{1,2},C_1}}$$

The encrypted packet is now sent to C_1 or C_2, which was waiting for the same this whole time. At this point, $AP_{1,2}$ waits for the component to send a `continue` message for it to send the public IV and AAD for the decryption.

Finally, the $AP_{1,2}$ sends the IV and AAD to C_1 or C_2 and waits for an acknowledgement of the finished transaction to be sent from the component:

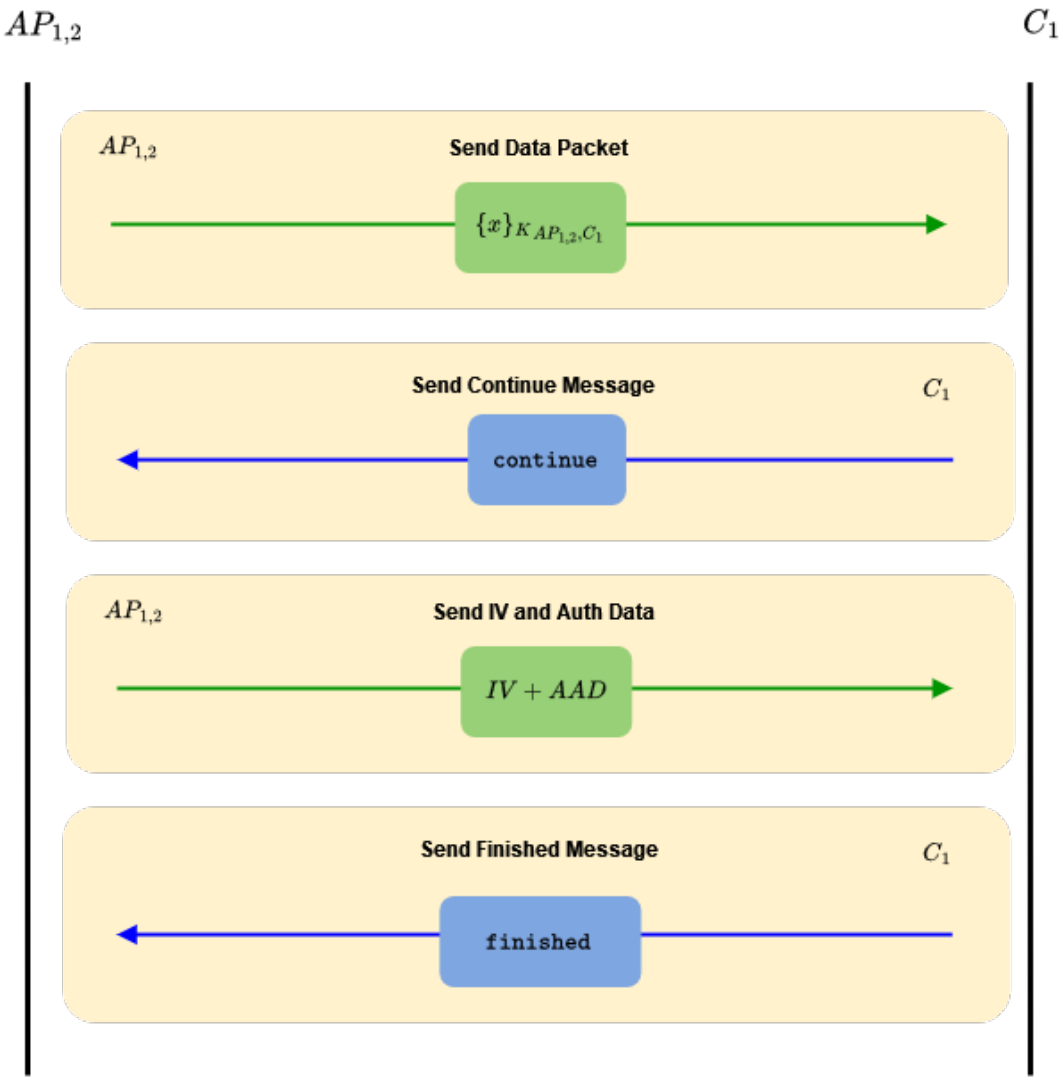

Figure 2: AP to Component Secure Communication Diagram

4.2.2 Component to AP

Because of the way I2C is implemented and $AP_{1,2}$ being the message controller, the direction $C_1 \rightarrow AP_{1,2}$ doesn't need a custom implementation of the `continue` and `finished` messages.

So, the communication goes the same way as section 4.2.1, but ignoring the communication flow messages:

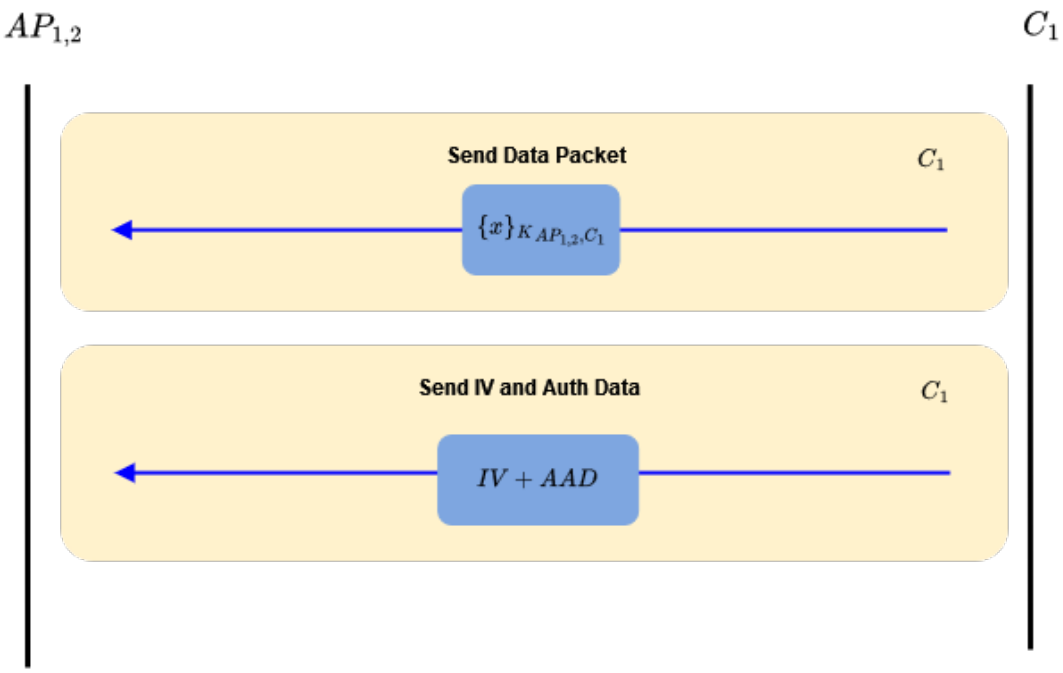

Figure 3: Component to AP Secure Communication Diagram

4.3 Attestation

The AP and component perform the full TLS handshake on successful PIN entry and exchange encrypted attestation data using the `secure_send` and `secure_receive` functions, which encrypt it and prevent it from being exposed.

We've prioritized security by integrating measures to prevent brute-force attacks and vulnerabilities by implementing a time delay mechanism. In doing so, the increased waiting period between attempts prevents the token to be brute-forced in a reasonable time.

Additionally, to address the threats to the side-channel analysis, we've implemented a constant-time compare function (`secure_cmp`). With this, we contain the risk of timing-based attacks.

4.4 Replacement

An AP is supposed to change the components it accepts when issued a replace command with a valid replacement token.

The entry of the token is is made secure through a constant time compare function (`secure_cmp`) along with delays that are increased on incorrect attempts to prevent against brute-force attacks.

5 Security Requirements

5.1 SR 1: Valid AP Boot

The application processor (AP) should only boot if all expected components are present and valid.

In our design, the AP only boots if it is able to safely execute the handshake, validating all attached components (Section 4.1.1). Since the certificate attests to the combination of ID_{C_1} and ID_{C_2} along with PK_{C_1}, PK_{C_2}, copying it to another component is ineffective unless SK_{C_1} or SK_{C_2} is leaked.

The SK_M is only in the global secrets (not accessible to attackers) and not loaded into any of the devices, so it is not possible to create a false certificate for another device.

Ownership of the keypair corresponding to the valid certificate is proved via a challenge-response mechanism, where the verifying device (the AP) sends a randomly generated nonce to the component (DH ephemeral key), receive a signed response and verify that the signature is valid (Section 4.1.1).

5.2 SR 2: Valid Component Boot

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

The components will only boot if they verify the application processor's certificate, and receive a valid signature for a randomly generated nonce for the private key corresponding to the keypair in the certificate (Section 4.1.1).

Similar mechanisms for security in SR1 apply here.

5.3 SR 3: Confidential PIN/Token

The Attestation PIN and Replacement Token should be kept confidential.

The PIN and Token are protected the following ways:

- Use a constant-time compare function to prevent timing analysis attack (`secure_cmp`)
- Use a delay before verifying the pin to make sure that it cannot be brute-forced in a reasonable timespan.

5.4 SR 4: Confidential Attestation Data

Component Attestation Data should be kept confidential.

To keep it confidential, the $AP_{1,2}$ validates the Attestation PIN and that the data will only be released if the PIN is correct.

Then, we ensure the handshake (Section 4.1.1) is done to validate the component and follow communications using `secure_send` and `secure_receive` (Sections 4.2.1 and 4.2.2).

5.5 SR 5: Post-Boot Integrity and Authenticity

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

The connection established using mutual authentication of certificates provides data integrity and authenticity, with added confidentiality.

All communication between boards are validated using the custom handshake (Section 4.1.1) and followed with the data transaction itself utilizing `chacha20_poly1305` on `secure_send` and `secure_receive` (Sections 4.2.1 and 4.2.2).