Final Design Document

ERA Team

25 February 2024

Contents

1 Introduction

Given an insecure design that meets the functional requirements, our team has designed a series of security solutions to fulfill the needs of the MISC device for protection. The following sections will go through each security solution we have proposed. The system given is divided into two essential parts:

- **Application Processor**: Master of the system, commanding and controlling components.
- **Components**: Slaves in the I2C communication protocol that execute commands coming from the AP.

Our job is to modify the source code of the insecure design to meet certain security requirements providing the AP and its components a secure environment for communication and data exchange.

2 Security Solution 1

2.1 Requirement

The Application Processor should not boot unless all components are present and valid to prevent any non desired boot.

2.2 Solution

To verify that each component is present, the AP should receive their IDs before doing anything. We can achieve this using the function `validate_components()`, which returns success only if all components respond with their IDs (indicating their presence)**(fig 1 and 2)**.

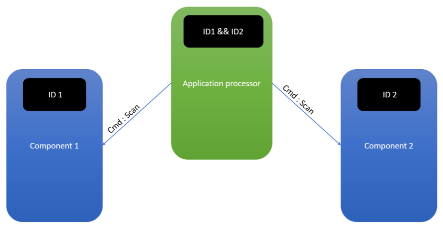

Figure 1: AP sending the scan command to both components

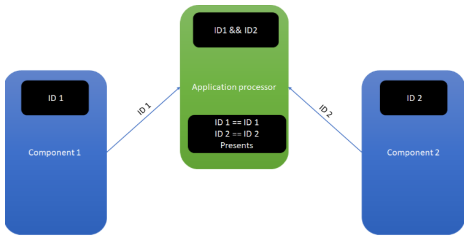

Figure 2: Both components respond with their IDs to mark their presence

Upon valid verification, the AP should request a validation message from each component, with each component responding with a confidential message to verify its identity. The AP should verify each component's secret_id before initiating the boot process. We proposed a hashing mechanism where a random common secret_id generated by a Python script (during the build) is concatenated with the component's ID (being the salt), then hashed using a standalone C function using the SHA_256 algorithm. Hence, this technique will give us a different secret message for each component known by the AP **see figures: 3 and 4**:

$$\text{Secret\_hash} = \text{hash}(\text{Secret\_id} + \text{salt}(\text{component\_ID})) \tag{1}$$

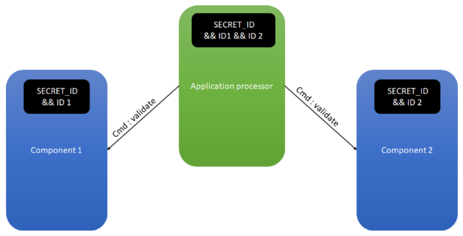

Figure 3: The AP sends the Validate command to both components

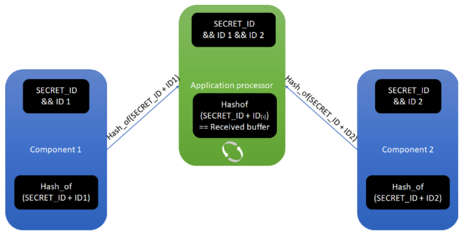

Figure 4: The components respond with a hash value of a secret_id and their respective component Id

2.3 Implementation

To implement this security solution, we propose adding another function to the application processor called `sign_component()` **(listing 1)**that verifies the signature (hash value) of each component. In fact, the component should send a valid signature (hash), which is verified against the received value. We introduce a process `process_sign`() **(listing 2)** where the function hashes the concatenation of the two values (COMPONENT_ID and COMPONENT_secret_id) and sends the result to the verification in order to achieve a secure and successful boot.

Listing 1: AP function that verify each component validity

```
int sign_components() {
if (validate_components()) {
     print_error("Components could not be validated\n");
     return ERROR_RETURN;}
     // Buffers for board link communication
uint8_t receive_buffer[MAX_I2C_MESSAGE_LEN];
uint8_t transmit_buffer[MAX_I2C_MESSAGE_LEN];

     // Send validate command to each component
for (unsigned i = 0; i < flash_status.component_cnt; i++) {
     // Set the I2C address of the component
    i2c_addr_t addr = component_id_to_i2c_addr(flash_status.component_ids[i]);

      // Create command message
command_message* command = (command_message*) transmit_buffer;
command->opcode = COMPONENT_CMD_SIGN;
char id2[50];
sprintf(id2,"%s",AP_secret_id);
char* hash2=hashme256(&id2);
sprintf((char*)command->params,"%s",hash2);

          // Send out command and receive result
int len = issue_cmd(addr, transmit_buffer, receive_buffer);
if (len == ERROR_RETURN) {
     print_error("Could not sign component\n");
     return ERROR_RETURN;
     }
char id[50];
sprintf(id,"%x",flash_status.component_ids[i]);
strcat(id,COMP_secret_id);
char* hash=hashme256(&id);

          // Check that the result is correct
if (strncmp((char*)receive_buffer,hash,65)) {
```

```
print_error("Component: 0x%08x didn't sign\n", flash_status.component_ids[i]);
    return ERROR_RETURN;
    }
}
    return SUCCESS_RETURN;
}
```

Listing 2: AP function that create a special ID for validity verification

```
void process_sign() {
    // The AP requested signature. Respond with a known hash value
    //creation of component identifier
    char id[50];
    sprintf(id,"%x",COMPONENT_ID);
    strcat(id,COMP_secret_id);
    char* hash=hashme256(&id);
    uint8_t len = sprintf((char*)transmit_buffer,"%s",hash) + 1;
    //sending signarure
    send_packet_and_ack(len, transmit_buffer);
}
```

3 Security Solution 2

3.1 Requirement

Components should not boot unless commanded by a valid AP.

3.2 Solution

Components verify the identity of the sender for each boot command (of type `COMPONENT_CMD_BOOT`) . To achieve this, we assign a random identifier to the AP (AP_secret_id) during each build phase using a Python script, storing it in `global_secrets.h` called from the Makefile. Additionally, we propose hashing this value, making it more difficult to reverse engineer. The AP, this time, should send the boot command along with a `SECRET_AP_ID` to the components before proceeding with the boot process.**fig 5 and 6**

3.3 Implementation

We used an already defined array (`command->params`) **(listing 3)** containing type `uint8_t` elements (which can be filled by characters) that accompanies each command to store AP credentials (secret hash to send). The component verifies this list every time a boot command is received and compares it to the value stored in global secrets**(listing 4)**, we enhanced also our design security by making component verify that array for each command.

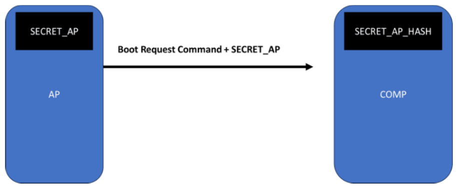

Figure 5: The AP sends a boot command to the component along with the hash value of the SECRET_AP

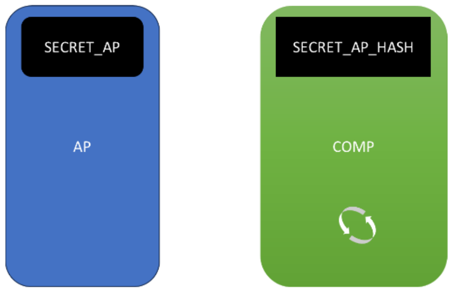

Figure 6: Upon a valid comparison the component begin the boot process

Listing 3: Filling params with AP credentials before sending commands

```
// Create command message
    command_message* command = (command_message*) transmit_buffer;
    command->opcode = COMPONENT_CMD_BOOT;
    char id[50];
    sprintf(id,"%s",AP_secret_id);
    char* hash=hashme256(&id);
    sprintf((char*)command->params,"%s",hash);
```

Listing 4: component part of code that verify AP's credentials or AP_secret_ID

```
         wait_and_receive_packet(receive_buffer);
         command_message* command0 = (command_message*) receive_buffer;

         //check credentials: we use this operation to prevent any external AP fr
    if (!strncmp((char*)command0->params,hash,32)){

         component_process_cmd();
         }
```

4 Security Solution 3

4.1 Requirement

AP_PIN and AP_TOKEN should be kept confidential.

4.2 Solution

After building the AP, AP_PIN and AP_TOKEN are registered in `ectf_params.h`, making them vulnerable.

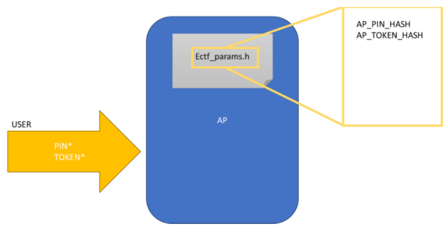

Figure 7: The user enters a PIN and a TOKEN value for attestation

Ectf_params.h
AP
Hashing...

Figure 8: the AP hashes the given AP and TOKEN to perform a comparison

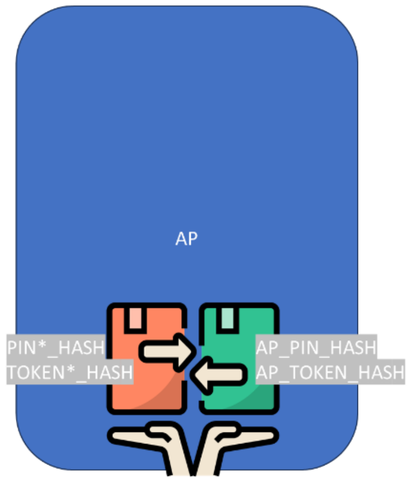

Figure 9: Depending on the comparison results the AP decides whether to to attest or not

5 Security Solution 4

5.1 Requirement

Attestation data should be kept confidential.

5.2 Solution

For the attestation data to be confidential, it should be stored encrypted once entered **(fig10)** using a secret key, we use for that an executable that encrypt **(fig 11)** and changes data from plain text to cipher one **(fig 12)**. Data should only be decrypted when a user enters a valid PIN and requests data from a valid component.

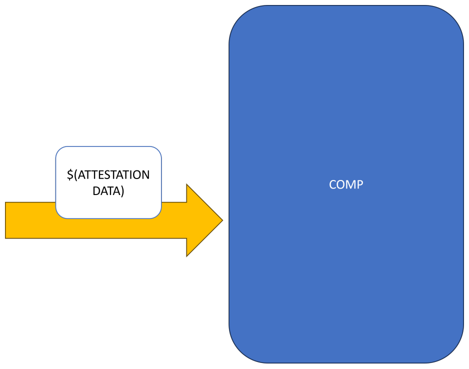

Figure 10: The user enters raw attestation data when building a component

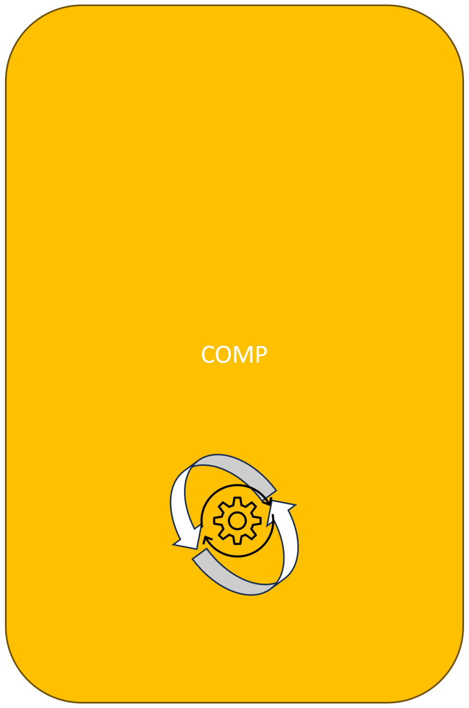

Figure 11: an executable takes care of encrypting this data just after the building process is done!

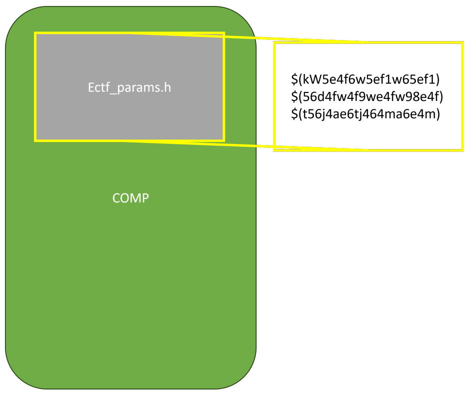

Figure 12: Ciphers are stored in the ectf_params header file

5.3 Implementation

During the build process, the Makefile (in the components folder) runs a C executable called **encrypt**, it ultimately accesses the attestation data in the *ectf_params* header file, retrieves it, encrypts it using a standalone AES encryption algorithm (ECB mod) with the hep of a 16 bit key generated previously in the global_secrets.h file, and then replaces the actual data for full confidentiality.

Upon calling the attest command by the AP, the component retrieves cipher data stored in the header file, decrypts it using the same key in the global_secrets file, and send the buffer (a well structured representation of the attestation data) to the AP for printing (only if the component demanded for its data is valid, and the PIN also)

6 Security Solution 5

6.1 Requirement

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

6.2 Solution

Authenticity can be guaranteed using a common secret or key exchange, ensuring that components and the AP only accept information after verifying the identity of each one. In our design authenticity and integrity of messages is preserved by dividing communication packets to three essential parts**(fig 13)** :

- **authenticity** this parts allows the receiver to verify the identity of sender by comparing a special hash value "hash of (AP_secret_id + COMPONENT_ID + COMP_secret_id)" sent along with the packet by the one known and calculated by the receiver.

- **integrity** it allows receiver to verify that the initial message didn't change and stays identical to the one received to prevent receiving any malicious code or getting any false information,it contain hash of the message concatenated by a secret generated the same way as previous ones using Python "hash of (Message+SECRET_messaging)".
- **Message** the message itself is sent in the same packet so that we ensure no one can modify it if it's sent separately.
- **End to End Encryption** Finally, we thought of encrypting the whole packets sent across the MISC whether it is sent from the AP or the components to further enhance security of messaging.

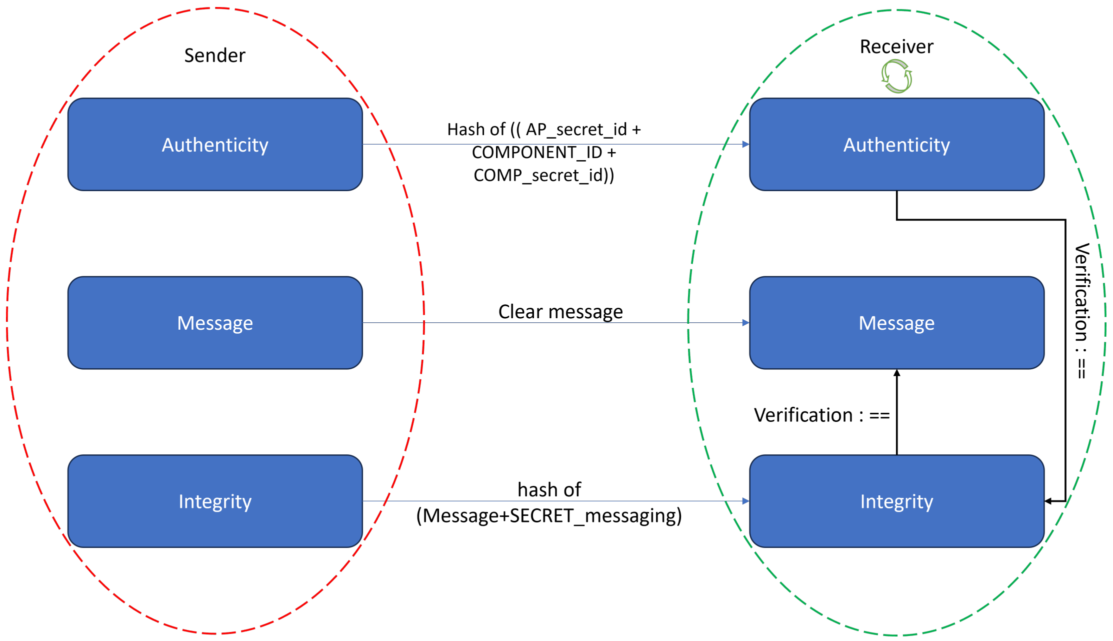

Figure 13: Ciphers are stored in the ectf_params header file

6.3 Implementation

Our solution is about creating a new message structure. We defined a `uint8_t` for each part of the message and implemented comparison between already known values and secrets and received ones for both parts (components and application processor). This implementation ensures both the authenticity and integrity of messages sent and received using the post-boot MISC secure communications functionality.

7 additional Security

7.1 Requirement

teams should think about probable attack scenarios and try to prevent them.

7.2 Solution

Our team dealt with four essential attack vectors:

- **BUFFER OVERFLOW**: We tried to change vulnerable functions by their secure alternatives like the fgets() instead of gets() and a custom comparison function instead of strncmp() **(fig14)**.

Figure 14: Buffer Overflow

- **BRUTE FORCE**:This attack can be applied on PIN and TOKEN because of their short lengths, In our design we added a timing factor for entering the PIN or TOKEN, We switched from 5s to 4.5s, 0.5s being widely sufficient **(fig15)**.

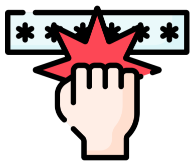

Figure 15: Brute Force

- **RAINBOW Attack**:Based on using rainbow tables of MD5 and SHA1 hash values and comparing them to the existing ones, an attacker can easily brute force a 6bytes buffer. Thus, We added in our design a random salt value to make it more complicated.
- **SIDE CHANNEL ATTACKS**: To protect our design from any form of side channel attacks, we designed our own comparison function that works in a non traditional way, performing high demanding power functions in a random manner each time a digit-to-digit comparison is performed making it hard for attackers to exploit it and get our secrets **(fig16)**.

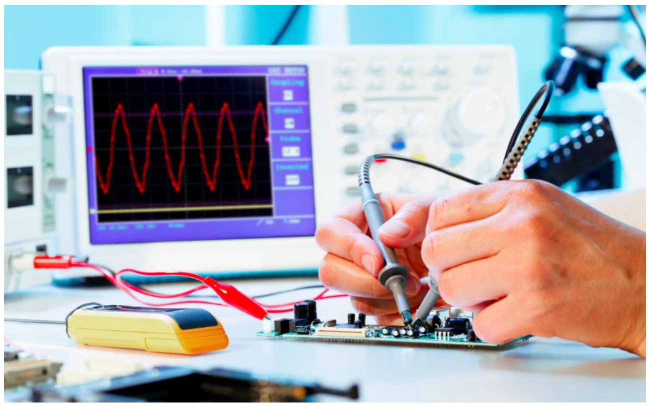

Figure 16: Side Channel Attacks

- **REVERSE ENGINEERING**: In order to protect our variable strings from being exposed, and consequently reverse engineered we thought of calling them the furthest away from where they're meant to be used.

7.3 Implementation

To implement these additional security requirements we changed the gets() used in recv_input() to fgets() adding a size_t parameter that defines the size of the buffer, we also switched strcmp() used in conducting the communication between the AP and the components with its secure version strncmp(), as well as for receiving PIN and TOKEN. A delay of 4.5s was added by our team after receiving a wrong PIN or TOKEN in the AP source code.

8 Conclusion

To conclude, we think we have diligently tackled the security vulnerabilities present in the MISC device by crafting a robust suite of solutions aimed at securing communication, authentication, and data integrity between the Application Processor and its components. Through our rigorous implementation of component validation, data encryption, and secured boot processes, we have significantly enhanced the device's defense mechanisms against unauthorized access and tampering. Additionally, our strategies to protect against prevalent attack vectors, such as buffer overflow, brute force, and side-channel attacks, underscore our comprehensive understanding of the security challenges and our proactive stance in fortifying the system. These measures not only fulfill the stipulated security requirements but also establish a strong foundation for a secure and reliable device ecosystem, ensuring that the MISC device operates safely in its intended settings.