Final Design Document
ECTF

IIT MADRAS
Indian Institute of Technology Madras

Contents

1 Build

1.1 Global secrets

We use a set of ten **master keys,** $(k_0, k_1, \cdots, k_9)$, which are all stored in every legitimate device from the manufacturer. These keys serve as pre-shared keys for our key-ratcheting mechanism (see subsection 3.2). For every communication between devices, one of the keys derived from these keys is chosen randomly.

Global secrets also contains the **flash magic** which is a set of bytes used to check whether the flash data retrieved on every boot is valid or not (see subsection 6.1).

1.2 `nop`-slide script

In order to make fault-injection attacks difficult to mount, random-length `nop` slides are added to the code at pre-defined locations at build time. This makes it impossible to know precisely which clock cycle to inject faults at, since the exact number of `nop`s is not known to the attacker unless the exact ELF is disassembled.

1.3 Hashing of credentials

Credentials like the attestation pin and replacement token are salted and hashed, and added to a separate header file, named `ectf_params_encrypted.h`. Only these values are used in the code. This makes it so that even if the hash is leaked from memory, the credentials still require brute-forcing to obtain.

2 Functional Requirements

2.1 List Components

2.1.1 Application Processor

In order to scan components, the application processor tries to brute-force all valid I2C addressees, by sending `COMPONENT_CMD_SCAN`, which is a pre-defined message of length 1. We use the function `insecure_handshake`, to send this message to each valid I2C address, and receive the response. If this fails, we know that a component is not present at that address.

2.1.2 Component

The component is an I2C slave, and so it blocks its execution until it receives commands from the application processor. When it receives a `COMPONENT_CMD_SCAN`, it replies with a `scan_message` containing its assigned ID.

2.2 Attest

2.2.1 Application Processor

When an attest command is received, the application processor calls the `attempt_attest` function. Here, the AP first checks if the component is already authenticated. If it is not, an authentication is done (see subsection 3.1). Now, a `COMPONENT_CMD_ATTEST` command is sent in an encrypted and authenticated manner to the component. The key ratcheting mechanism described in subsection 3.2 ensures that a different key is used for each message, and so, messages cannot be replayed.

2.2.2 Component

An unauthenticated component first waits for a handshake from the AP. Once authenticated, it listens for messages. When it securely receives a `COMPONENT_CMD_ATTEST` command, it securely replies with the stored attestation data.

2.3 Replace

The AP stores its list of provisioned components in flash memory. On receiving the replacement token through the host tools, take component IDs to swap are taken, and the flash is rewritten with the new component IDs. Please refer subsection 5.3 for the security aspects of the implementation.

2.4 Boot

Components that are not authenticated by the AP are authenticated with secure handshakes (subsection 3.1), and a `COMPONENT_CMD_BOOT` is sent securely to all the components.

2.5 Secure Send & Receive

The communication protocol mentioned in section 3 is used for secure communications for both pre-boot and post-boot messages.

3 Communication Protocol

3.1 Authentication

Both parties involved in a communication first authenticate each other according to the following scheme.

1. The AP randomly picks a key k_i $(0 \leq i \leq 9)$ out of the set of keys stored in it, and generates a random value r_1, of length 32 bytes. i and r_1 are sent to the component.

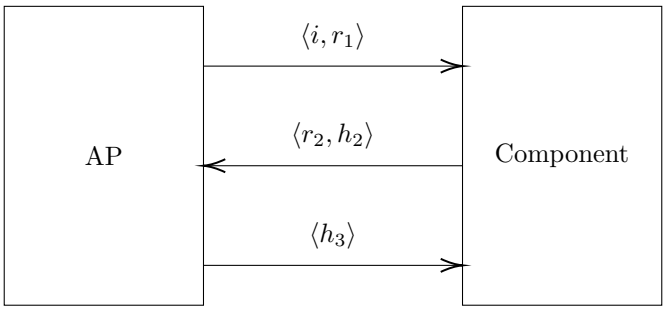

Figure 1: Three-way handshake

2. The component sends back another random 32-byte value r_2, and also sends the SHA3-HMAC, $h_2 = \text{HMAC}_{k_i}(r_1|r_2|\text{ID})$, where ID is the component ID. The AP validates this HMAC, and authenticates the component.

3. The AP sends back the keyed hash $h_3 = \text{HMAC}_{k_i}(h_2)$. By validating this HMAC, the component authenticates the AP.

4. Keys are derived (see subsection 3.2) from each of the ten pre-shared keys using h_3 as the nonce. Every channel between two components has its own associated set of derived keys.

3.2 Key-ratcheting mechanism

In order to prevent replay attacks, and to make power analysis attacks of AES impossible, a key-ratcheting mechanism is used to ensure that the same key is never used twice for encryption.

Initially, every key k_i is initialized using a nonce h, by XOR-ing it with the nonce, to derive a round-zero key $K_i^0 \equiv k_i \oplus h$. In order to advance a key by a round, we perform a SHA3-HMAC, as follows:

$$K_i^{j+1} = \text{HMAC}_{K_i^j}(\underbrace{j|j|\cdots|j}_{32 \text{ bytes}})$$

After every use of a key, it is advanced by one round. This ensures that the same key is never used for two different encryptions.

3.3 Secure communications

The key-ratcheting mechanism described in subsection 3.2 is used for sending messages in an authenticated manner.

In order to send a message m, a key is chosen randomly from the set of derived keys. This key, K_i^j is used to encrypt m using AES. The AES-CBC mode is used, with a randomized initialization vector iv generated on each encryption. This vector is appended to the start of the ciphertext. The ID

of the component involved in the conversation is appended to the start of the message, before it is encrypted. Thus, the ciphertext for m is given by:

$$C \equiv (iv)|\text{AES-CBC}_{K_i^j}(iv, \text{ID}|m)$$

Next, SHA3-HMAC of the ciphertext is calculated using the same key. This gives us a hash, given by:

$$H \equiv \text{HMAC}_{K_i^j}(C)$$

Now, the final message is sent out as:

$$i|j|C|H$$

The key is then advanced to round $j + 1$.

The receiving device advances its key to the round specified, and authenticates the sender by verifying the HMAC. The ciphertext is then decrypted and then processed. As a security measure, the receiver's key is not allowed to advance forward by more than two rounds. After decryption, the receiver's key is advanced to round $j + 1$.

3.4 I^{2}C Implementation

The I^{2}C implementation is based on the base design with some added security features. All the software registers are implemented as secure buffers to prevent any buffer overflows. The application processor drives the communication and the components handle the communication with an interrupt service routine. Checks have been emplaced to make sure that devices overflow the software registers. Every transmission accesses three software registers through I2C messages – the length of transmission, the transmitted packet, and an indicator for marking the end of a transmission.

A software register named `READY` is used to indicate if a component is actively blocking and waiting for a transmission. The AP polls for a component to indicate readiness before it transmits a message. After every transmission, the AP waits till the transmission is acknowledged explicitly. The application processor has timeouts in the form of poll limits for each send and receive. If the component does not respond to the poll, then we set the board to the error state using `panic()`. The component usually polls infinitely for any send or receive.

4 Custom libraries

A brief summary of the functions and structs used in the `application_processor.c` and `component.c` files is given below.

4.1 Secure buffer

Rather than using pointers directly for buffers, a wrapper struct named `buf_u8` is used. This keeps track of the size of each buffer, and provides memory-safe operations that call `panic()` when the size constraints are exceeded.

4.2 Cryptographic functions

A `crypto_config` struct is used to store the component ID and set of derived keys for each communication channel between the AP and a component. The `ap_auth` and `component_auth` functions initialize this struct. The `ap_send`, `ap_receive`, `component_send`, and `component_receive` functions are used for sending and receiving messages in the AP and components.

4.3 I2C functions

4.3.1 Application Processor

`send_packet_and_ack` polls a component's `READY` register until it indicates that it is ready to read a message. It then writes a buffer over I2C, and then polls the component until it has acknowledged the message.

`poll_and_receive_packet` polls a component until it has a message to send. Once a component has written a message into its `TRANSMIT` register and marked it in `TRANSMIT_DONE`, this packet is read out, and acknowledges it.

4.3.2 Component

`send_packet_and_ack` writes a buffer to the `TRANSMIT` register, and marks this in `TRANSMIT_DONE`. The component spins indefinitely until the AP reads this register, and acknowledges that it has received the message.

`wait_and_receive_packet` first sets the `READY` state of the component, and then spins until the AP writes a message to the component. Once the AP is done writing, the component stops spinning, and unsets `READY`.

5 Security Requirements

5.1 Security Requirement 1

We use the three-way messaging scheme mentioned in subsection 3.1 to authenticate devices in the system.

5.2 Security Requirement 2

Components boot only when they receive an authenticated message via the secure messaging scheme given in section 3. This message is sent out only if all provisioned components are authenticated successfully.

5.3 Security Requirement 3

The pin and token is not being written to the flash of the application processor, instead their SHA3_256 hash is written. When the host tools send these to the processor for authentication, the values are hashed and the hashes are compared in a constant-time manner in the application processor. This protects against attempts to dump flash through leaks.

5.4 Security Requirement 4

Patient details are stored in component code memory and only shared upon request from an authenticated master and any encrypted communication which is not consistent with the hashes are considered malicious and dropped.

5.5 Security Requirement 5

All post-boot communications take place using the secure messaging scheme described in subsection 3.3, to guarantee confidentiality, integrity, and protection from replay attacks.

6 Additional security measures

6.1 Flash

Flash is used in our implementation for following purposes

1. Storing provisioned components.
2. Storing information about any malicious activity in order to make sure simple reset doesn't bypass the cool-down enforced.

In order to ensure that flash data is not corrupted, a set of magic bytes is written to flash. Only if this is read correctly, will the flash be treated as valid.

6.2 Rate-limiting

Upon detecting any malicious activity or failure, the system goes to a panic mode which needs appropriate cool down (5 sec) before it starts functioning again properly. This ensures that brute-forcing is impractical.

6.3 Random delays

Random delays are added to various parts of the code using the `spin()` function. This ensures that fault injections are difficult to mount.

6.4 Timing attacks

1. WolfSSL's hardening options are used to make sure cryptographic algorithms used in the setup is side channel resistant.
2. All comparisons are made constant time comparisons to avoid timing attacks
3. SHA3 hashing algorithm is used since its resistant against side channels upto date.

6.5 Memory safety

Every buffer handled in the code is abstracted into a struct which maintains the size of the buffer alongside it. Any operations done on the buffer are subjected to various bounds checks to avoid buffer overflows.

6.6 RISC-V co-processor

The RISC-V co-processor is disabled on every reset, to ensure that no attacks relating to it can be mounted onto a microcontroller.