Medical Infrastructure Supply Chain Design Documentation

2024 MITRE eCTF Initial Design

Date: 19 March 2024

Written By: United States Air Force Academy (Code Monkeys🐒)

SR 1:

In the final design of SR 1, we implemented a global secret key. With the application process as an I2C controller, a new command was created called COMPONENT_CMD_VERIFICATION within the component_cmd_t struct. Then, with this new command, a new function in the application_processor.c called key_sync(). When the attempt boot command is called from the user, before the boot function is called, the key_sync function is called first. The key_sync function is shown in Figure 1.

```
207 int key_sync() {
208
209     // Buffers for board link communication
210     uint8_t receive_buffer[MAX_I2C_MESSAGE_LEN];
211     uint8_t transmit_buffer[MAX_I2C_MESSAGE_LEN];
212
213     // Send key verification command to each component
214     for (unsigned i = 0; i < flash_status.component_cnt; i++) {
215         // Set the I2C address of the component
216         i2c_addr_t addr = component_id_to_i2c_addr(flash_status.component_ids[i]);
217
218         // Create command message
219         command_message *command = (command_message *)transmit_buffer;
220         command->opcode = COMPONENT_CMD_VERIFICATION;
221         //command->params[0] = globalKey;
222
223         // Send out command and receieve result
224         int len = issue_cmd(addr, transmit_buffer, receive_buffer);
225         if (len == ERROR_RETURN) {
226             print_error("Could not validate key\n");
227             return ERROR_RETURN;
228         }
229
230         verification_message *verification = (verification_message *)receive_buffer;
231
232         // Check that the result is correct
233         if (verification->component_id != flash_status.component_ids[i] || verification->verification_key != verificationKey)
234             print_error("Verification with Component ID: 0x%08x was invalid\n",
235                         flash_status.component_ids[i]);
236             return ERROR_RETURN;
237         }
238     }
239     return SUCCESS_RETURN;
240
241 }
```

Figure 1: key_sync function

Within the key_sync function we combined sections of code from other functions. We first define local variables receive_buffer and transmit_buffer, which are buffers that contain packaged send and receive data. These buffers are used between the AP and the components. Then, we within the function, we verify each component. To do this, we first set the I2C address of the specific component that we want to verify. Then, we create a command message, casting

the transmit buffer to a command message type so that we can input commands. We modified the component_cmd_t enumerated type definition to include a COMPONENT_CMD_VERIFICATION command in both the component.c and application_processor.c files. We then insert this command into the transmit buffer. Finally, we issue the full command to the component. Once the command is sent, we check for transmit errors. Within the component.c file, we implemented a function called process_key_sync, shown in Figure 2.

```
227  void process_key_sync() {
228      verification_message* packet = (verification_message*) transmit_buffer;
229      packet->component_id = COMPONENT_ID;
230      packet->verification_key = verificationKey;
231      send_packet_and_ack(sizeof(verification_message), transmit_buffer);
232  }
```

Figure 2: process_key_sync function

Also, within component_process_cmd, we added an additional case which allows us to run the process_key_sync function based on the COMPONENT_CMD_VERIFICATION command. Within the process_key_sync function, the transmit buffer (on the component side) is cast into a verification message. This is a new message that we created specifically for SR1 and SR2 implementation. The new message is shown in Figure 3.

```
63  typedef struct {
64      uint32_t component_id;
65      uint32_t verification_key;
66  } verification_message;
```

Figure 3: verification_message struct

Within this struct, a verification_key is added, so that the component can verify with the with application processor. With the verification message cast into the transmit buffer, the component is able to transmit a verfication_message back to the application processor. First, the component

ID is entered into the verification message. Then, a verification key is also entered into the verification message. This key is randomly generated and is located in the global secrets file. Once these two fields are loaded into the transmit buffer, the verification_message is then sent back to the application processor. Then, returning to the key_sync function in application_processor.c, the receive buffer in the AP is cast to a verification_message. The contents of the received verification_message are then checked with the the flash memory and the global verification key accessed on the AP side. After successful validation of the global secret from all of the components, the AP will be able to boot. If either of the received component IDs or the verification key does not match, process_key will return an error and the AP will not boot. This is shown below in Figure 4, the attempt_boot function in application processor.c.

```
434  void attempt_boot() {
435      if (key_sync()) {
436          print_error("Components could not be validated\n");
437          return;
438      }
439      print_debug("All Components validated\n");
440      if (boot_components()) {
441          print_error("Failed to boot all components\n");
442          return;
443      }
444      // Print boot message
445      // This always needs to be printed when booting
446      print_info("AP>%s\n", AP_BOOT_MSG);
447      print_success("Boot\n");
448      // Boot
449      boot();
450
451  }
```

Figure 4: attempt_boot function

The first check in this function is the key_sync function. If the key_sync returns an error, the function will return and nothing else from the function will be run. This includes the boot command on line 449, which would boot the AP.

SR 2:

In the final design of SR 2, we used the same global secret key used for SR 1. The COMPONENT_CMD_VERIFICATION within the component_cmd_t struct is used again for SR 2. The idea behind this is that key_sync will return an error if there is either a bad component or a bad AP, as the verification key is a global variable shared by both the AP and the component. As mentioned before, in attempt_boot, the key_sync function is called first, which in turn calls the process_key_sync function in component.c (see above). This time, while looking at Figure 4, it can be seen that the boot_components command on line 440 will also not be called if key_sync does not return. Thus, the components cannot boot if there the keys cannot be synced between the AP and the components. We specifically simplified this functionality by using only a single function because we knew that the establishment of a secret global verification key would allow us to use two-way verification.

SR 3:

In the final design of SR3, we attempted to implement delay of 4-5 seconds each time the wrong Attestation PIN or Replacement Token is entered in order to slow the attackers in a brute force attack. The delay would be randomized each time the wrong Attestation PIN or Replacement Token is entered in order to prevent a side channel attack. Unfortunately, we were not able to fully implement this design, as the delays that we used were too long, which did not meet the timing requirements set forth by the competition.

SR 4:

After checking if the flash is unwritten, the attestation data of location, data, and customer from ecft_params.h are stored in the flash memory using flash_simple_write() on a boot. The flash address where the attestation data will be stored is randomized. In the range of flash addresses from 0x10045FFF to 0x1007E000, the address of

MXC_FLASH_MEM_BASE + ${random_numberTwo}*MXC_FLASH_PAGE_SIZE

is generated. The available flash address blocks are from the 23rd block of flash memory to the 64th block of the flash memory. Therefore, a random number is generated within the range of 23 to 64 to select a random block of flash memory to store the attestation data. This attestation data stored in the flash memory will only be returned by AP for a valid Component if the user provides a valid Attestation Pin. In doing so, the attestation data is kept confidential.

SR 5:

In the final design of SR 5, we attempted to utilize the AES128 encryption functions from WolfSSL to encrypt and decrypt the data being sent between the AP and Components inside the secure_send and secure_receive functions. A global secret key was created to perform this symmetric encryption/decryption. When utilizing any of these functions, the data is encrypted before being transmitted by either the AP and Component and encrypted once received by the other device. We were not able to pass post-boot testing with the slack testing service, so we commented out our attempts to use these functions as we could not figure out what was wrong with our implementation.

However, we implemented Exclusive OR encryption, which is a symmetric encryption used for operation speed and simplicity due to time constraints. The XOR encryption takes each uint8 data to be sent and received through secure_send and secure_receive and XOR with MESSAGE_KEY – a secret key created from the bash script. This is shown in Figure 5.

```
## key for SR 5 for encrypting I2C message
random_char=$(LC_CTYPE=C tr -dc 'a-zA-Z0-9' < /dev/urandom | head -c1) ## random char, ensure a single byte character

echo "#define MESSAGE_KEY '${random_char}'" >> global_secrets.h
```

Figure 5: Bash script for generating secret key

The bash script creates a random char out of all the single-byte possible chars and stores the character as MESSAGE_KEY. Single-byte possible chars include a through z, A through Z, and 0 through 9. We made the MESSAGE_KEY randomized and stored in global secrets so that encryption and decryption systems are more secured. Within the secure_send and secure_receive functions, the encryption and decryption go through each of uint8 elements of buffer array and performs an XOR operation (^) to the data with MESSAGE_KEY. This is shown in Figure 6.

```
int secure_receive(uint8_t* buffer) {

    uint8_t lengthReceived = wait_and_receive_packet(buffer);

    for (int i = 0; i < lengthReceived; i++) {
        buffer[i] = buffer[i] ^ MESSAGE_KEY;
    }

    return lengthReceived;
}
```

Figure 6: secure_recieved function

The for loop goes through each element of the buffer array until the length, which is passed in as a parameter or a variable found. The encryption and decryption processes are the same by performing XOR operations between the data and the key.