Michigan Technological University
eCTF Final Design Document

Design Summary

We based our implementation heavily on the reference code provided by MITRE. All of our code is still in C. This portion of the document explains the security features we built on top of the reference design.

Key Storage

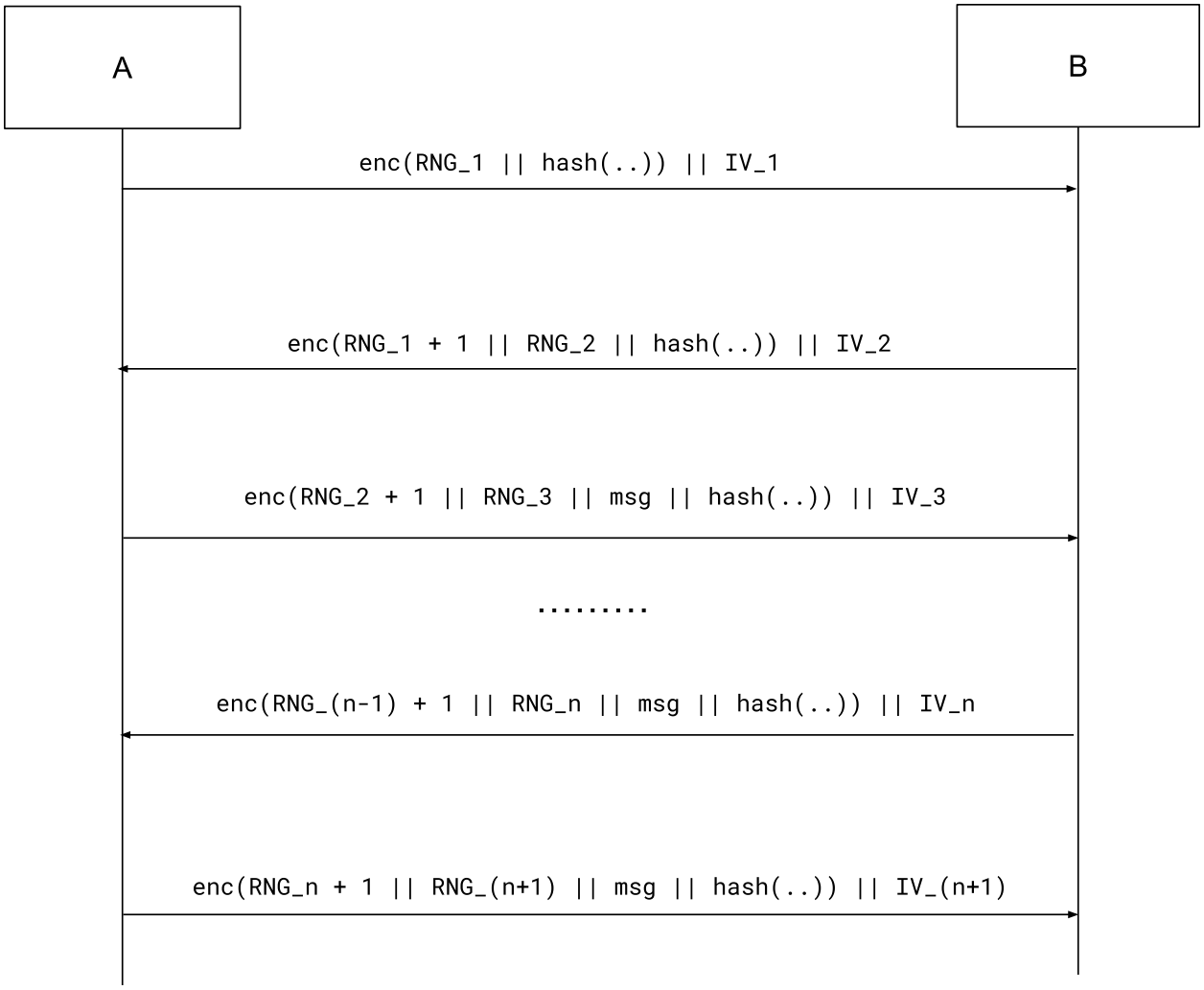
A random 128-bit encryption key is generated by `deployment/Makefile` and embedded into `deployment/global_secrets.h` when the system is built. This key is shared between the AP and all provisioned s for that deployment, and used as an AES key to secure communications. The key is generated by the `Makefile` by reading from `/dev/random` as shown in the code snippet below:

```
all:
# Generate a random encryption key for use between AP and Component
   echo -n "#define KEY {" > global_secrets.h
# od prints 16 bytes of hex in format: 00 11 22 ... ff
# use sed to add commas and 0x prefix: , 0x00, 0x11, 0x22 ... 0xff
# use cut to remove the extra comma from the front: 0x00, 0x11, 0x22 ... 0xff
# use head to remove the trailing newline from the output
   od -An -tx1 -N16 /dev/random | sed "s/ /, 0x/g" | cut -b3- | head -c-1 >>
global_secrets.h
   echo "}" >> global_secrets.h
```

A compromise of this key would result in the full compromise of our system, but that is an inevitable risk of any compile-time secret. A public-key implementation where symmetric keys are negotiated still requires the private keys to be kept secret. Our key should not be accessible unless the packaged binary code is decrypted, which is out of scope for the attackers.

Cryptographic Handshake

Whenever secure communication is needed between the AP and a component, the messages will be sent over I2C in the format shown by the diagram below. This protocol is similar to a TCP three-way handshake, but utilizes encryption and hashing to also provide confidentiality and integrity to the exchanged messages. Here, "`A`" is the party initiating the communication and "`B`" is the party receiving the first message. The "`||`" operator represents concatenation. `hash(..)` represents the hash of all other plaintext inputs to the `enc()` encryption function. The `IV`s are initialization vectors used for encryption and decryption, intentionally left unencrypted. From the third transmission onward, `msg` can be any data that the parties wish to send to one another.



Some more notes on the handshake protocol:

- After the second transmission is received, A believes that B knows the secret encryption key
- After the third transmission is received, B believes that A knows the secret encryption key
    - Now both sides of the communication can trust the messages sent and received
- The chain can be extended for any number of transmissions, but three is the minimum needed for confidentiality and integrity in the communications
- The hash is encrypted to avoid chosen-ciphertext style attacks that could be induced by changing the initialization vector
    - If an attacker received transmissions with an unencrypted hash, they could feasibly recover the RNG values, change the IV, and successfully predict the new hash value, creating a slightly forged message that appears valid to the other party

Cryptographic Handshake Implementation

There are some slight deviations from the diagram in the actual code that we use to implement our cryptographic protocol. The struct declaration for our messaging type is in the below code snippet:

```c
#define MAX_CONTENTS_LEN 198
#define HASH_LEN 32
#define IV_LEN 16
// Amount of struct, starting from byte 0, that is encrypted
// This means we encrypt from rng_chal up through byte 17 of hash
#define ENC_LEN 224

/*
  The total size of this struct must be exactly 255 bytes to fit into one
  I2C message. Use the pragma pack compiler directive so that the compiler
  does not insert padding between fields, which would mess up serialization
*/
#pragma pack(push,1)
typedef struct msg_t {
   // RNG challenge / response values for cryptographic handshake
   uint32_t rng_chal;
   uint32_t rng_resp;
   // Reusing this field from reference design for simplicity
   uint8_t opcode;
   // contents are unencrypted when set by user, encrypted when sent to I2C
   uint8_t contents[MAX_CONTENTS_LEN];
   // Hash of rng_chal through contents, partially encrypted
   uint8_t hash[HASH_LEN];
   // IV used for encryption, always in plaintext
   uint8_t iv[IV_LEN];
} msg_t;
#pragma pack(pop)
```

The main difference is that we only encrypt the first 17 bytes of the hash in our implementation. This is because our AES CBC encryption function does not support padding, so it operates on messages that exactly fit into the 16-byte block boundaries. If we added another block, we would end up encrypting over part of our initialization vector, which would break the functionality of the messaging. However, this still leaves 2^{136} bits of entropy in the encrypted portion of the hash, so it is entirely infeasible for an attacker to forge a valid hash for any modifications they make to the message.

Random Number Generation

We perform our random number generation using the built-in TRNG module on the MAX7800 FTHR chips. This is implemented in our `general_util.c` file, using code that is very similar to the MSDK examples for the chip.

Cryptography Implementation

We use WolfSSL to implement our encryption, decryption, and hashing functions. Symmetric key encryption and decryption for the secure messaging protocol use 128-bit AES in CBC mode. We use CBC mode so that the entropy from random initialization vectors and RNG challenge values propagate throughout the ciphertext, which makes replay attacks infeasible. The hash function uses SHA256 to hash data. We formatted our cryptographic code based off of the examples provided in the official WolfSSL documentation.

Timing Attack Resistance

Both the Attestation PIN and Replacement Token checks are vulnerable to timing side-channel attacks in the reference design. We implement two layers of security to avoid this side channel.

The first is an implementation of `memcmp` that runs in constant time for a fixed comparison size, and does not leak the 'less-than' or 'greater-than' information, only a 'success' or 'failure' return value:

```
int secure_memcmp(uint8_t *a, uint8_t *b, size_t len)
{
   uint8_t cmp_status = 0;

   for(int i = 0; i < len; i++){
       if(a[i] != b[i])
           cmp_status = 1;
   }

   return cmp_status;
}
```

The second is a randomized delay right before checking either the PIN or Token. We delay for between 0.5 and 1.5 seconds between PIN/Token entry and validation to further throw off any side-channel analysis:

```
time_delay(500000, 1500000);
   if ((strlen(buf) == 6) && !secure_memcmp((uint8_t*) buf, (uint8_t*) pin, 6)) {
       print_debug("Pin Accepted!\n");
       return SUCCESS_RETURN;
   }
   print_error("Invalid PIN!\n");
   return ERROR_RETURN;
```

Delays

To deter brute-forcing of either the Attestation PIN or Replacement Token, we delay the AP for 4 seconds on an incorrect PIN/Token entry before accepting another command from the user.

Flash Protection

The last page of flash memory on the AP holds the number of provisioned components and their IDs. If an attacker could change the values in this portion of flash, they could potentially cause the AP to misbehave. To protect against this, we keep that data encrypted and hashed when it is on the flash memory, using the following struct in our code:

```
// Datatype for information stored in flash
#pragma pack(push,1)
typedef struct {
   uint32_t flash_magic;
   uint32_t component_cnt;
   uint32_t component_ids[32];
   uint8_t  hash[HASH_LEN];
   uint8_t  iv[IV_LEN];
} flash_entry;
#pragma pack(pop)
```

Here, `flash_magic` is generated in a similar way to the global AES key by a `Makefile`. We encrypt from `flash_magic` through the first 24 bytes of `hash` for the same reasoning given for encrypting a portion of the hash in our cryptographic handshake protocol. This maintains the integrity of the data stored on flash. If either the magic number or hash fails to validate on AP boot, the AP will rewrite flash with the originally-provisioned component information.

Buffer Overflow Resistance

To communicate over serial with a user of the device, we utilize the provided `host_messaging.h/.c` code from the reference design. However, this code utilizes `gets()` to read input from the user, which is extremely insecure since it does not restrict the length of the input it copies into a buffer, leading to buffer overflow attacks that could compromise the entire system. To avoid this, our code uses `fgets()` instead of `gets()`, which allows us to cap the length of a received input line to a safe value.

Stretch Goals

Our initial design document mentioned anti-glitching protocols as a wishlist / stretch goal item, but we were unable to complete that part in time for the attack phase.

Security Requirements

This section of the document justifies how our implementation of the MISC satisfies the five security requirements.

Requirement 1: AP Boot

“The Application Processor (AP) should only boot if all expected Components are present and valid.”

When the AP receives the boot command, our implementation performs a cryptographic handshake with all provisioned Components to ensure that they are on the I2C bus and valid – that is, they have knowledge of our secret encryption key. Once the validity is confirmed, the AP will instruct all Components to boot through another message continuing the handshake protocol. If all Components succeed in booting, the AP will finally boot itself. If any provisioned Component is missing or fails to pass the cryptographic handshake check, the AP will not boot.

Requirement 2: Component Boot

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

As described above, the AP starts the boot process by validating all provisioned Components. Only after a full validation does the AP tell each Component that it is okay to boot. If the first Component passes validation but the second Component fails, the AP will tell the first Component to abort the boot process. Of course, the Component is also implementing the cryptographic handshake to validate that the AP knows the secret encryption key. If the AP sends a malformed message at any point in the validation and boot message sequence, the Component will refuse to boot.

Requirement 3: Attestation PIN and Replacement Token

“The Attestation PIN and Replacement Token should be kept confidential.”

The PIN and Token are only present in the code through the `ectf_params.h` file for the AP, so they cannot be recovered from the code without breaking the compiled binaries’ encryption, which is out of scope. Additionally, they cannot be feasibly brute-forced or recovered through a timing side-channel through the timing attack resistance and delay functionality that we explained in the prior section.

Requirement 4: Component Attestation Data

“Component Attestation Data should be kept confidential. Attestation Data should only be returned by the AP for a valid Component if the user is able to provide the correct Attestation PIN.”

Securing requirement 3 ensures that our AP implementation will only start the attestation process if a user provides the correct PIN. Additionally, our AP implementation checks that the requested Component ID was actually provisioned for the system. Then, the AP will begin a cryptographic handshake with the Component. The first three transmissions contain no data aside from the RNG challenges, hashes, and initialization vectors. If the handshake passes, both the Component and AP know that the other is valid. Then, in the fourth and final transmission, the Component will send the AP its Attestation Data, which will then be displayed to the user by the AP. If any portion of the cryptographic handshake fails, the AP or Component will abort the process and not return the Attestation Data.

Requirement 5: Post-Boot Message Integrity

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

We implement the post-boot messaging using our cryptographic handshake protocol. This has the side effect of also providing confidentiality, so it’s a bit overkill but the choice made the overall implementation easier. As described earlier, the encrypted hash portion of our handshake protocol provides message integrity. Authenticity is provided by the fact that parties need knowledge of the secret encryption key to create and read valid messages. In our post-boot functions, the party that is sending the message initiates the handshake, and sends the message on the third transmission only if the previous two transmissions were successful and valid. The receiving side again checks that the message’s hash and RNG response are valid, and if so accepts the message and copies it into the receive buffer. Copying is capped at 64 bytes to avoid any sort of buffer overflow here. If either side detects a malformed message at any point during the handshake, it will abort its secure communication function with an error return.