Table of Contents

Introduction

Our Team

Our team is comprised of individuals from the University of Nebraska-Omaha from Computer Science and Cybersecurity backgrounds. Each member of our team brings diverse experience to the table and is assigned to distinct roles. The roles of each member can be seen below.

Arber Salihu	Project Lead | Task Manager | Secure Designer | Developer
Jake Braddy	Project Lead | Task Manager | Secure Designer | Developer
Md Monirul Islam	Lead Developer
Joe Clarke	Lead Developer
Dominic Fate	Secure Designer | Developer
Alex Matzar	Developer
Abdoul Latoundji	Developer
Logan Mears	Documenter

NULLify

A majority of our team are individuals from NULLify, the University of Nebraska-Omaha's student-led computer security club. Founded in 2011 and currently advised by Dr. Bill Mahoney, NULLify aims to provide a platform for members of the community to discuss information security topics, learn skills that employers require of information security professionals, and prepare for capture the flag information security competitions. NULLify has competed in numerous capture the flag competitions ranging from international to local competitions. The club also reaches out to the community through giving talks at various workshops and events.

Design Phase

Initial Logic of Design

Figure 1 includes our general idea of how the two components and application processor interact with each other. This diagram also illustrates the time frame in which the wake and boot process take place.

Starting from the left side of the diagram, the Host PC supplies both components and the application processor with power. The dotted line around the AP and components illustrates the wake time of 1 second.

In the next step of the diagram, the AP checks if the components exist. If they do, we continue to the next step. If the components do not exist, a reboot is required. Next, AP and components check their respective integrities using a hash of the known-good firmware. If the integrity of the AP and components is valid, we continue to the next step. If the integrity of the AP and components are not valid, a reboot is required. Then the devices will open an encrypted communication channel which utilizes information from the shared secret. Finally, the devices will utilize challenge/response functionality to determine the authenticity of each other (the AP will query the components and the components will query the AP). The contents will have varying timing constraints (i.e. ping must happen within 0.5 seconds, or it will be considered a failure) to ensure the boot phase of happens within the 3 second limit.

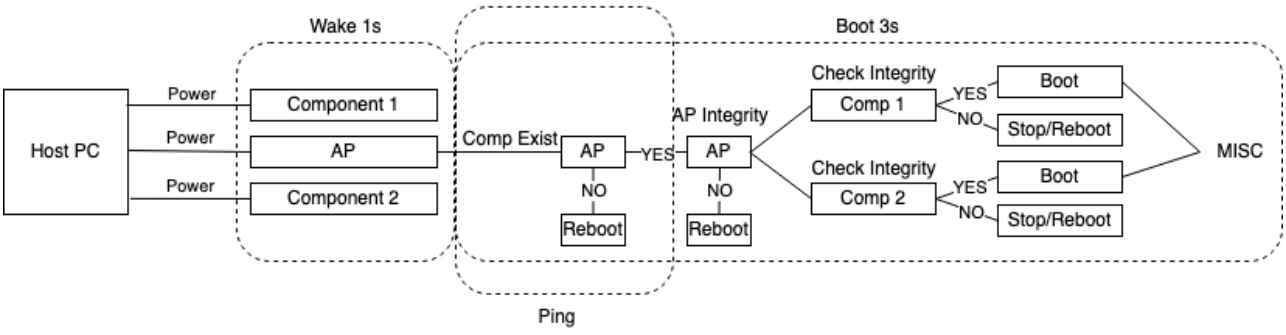

Figure 1: *Logic of Design*

Security Requirements

This section includes details on the steps we planned to take to accomplish the security requirements outlined by MITRE.

Requirement 1

The AP should only boot if all expected Components are present and valid.

These are the steps taken to address security requirement 1:

- To check if the components are present, we will be sending a ping to them via I2C communication and waiting for a response. If there is a response, the components are present and the boot process will continue, if no response is received within the given time limit, then the MISC will reboot.

These are the steps that were not implemented in our final design:

- To check the integrity of the Component, we are planning to do a challenge/response methodology which will rely on a randomized selection of predetermined questions and answers. These questions will consist of problems like “what is the hash for this section of code?” The answers will need to be provided in the same order they were asked.
- Until the challenge response is solved, the component will not boot, and if the challenge response fails, the MISC will be rebooted. If the time constraint is validated, the MISC will be rebooted.

Requirement 2

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

These are the steps taken to address security requirement 2:

- To secure the AP, we will be implementing a secure boot function to check the integrity of itself before it boots. This secure boot functionality relies on a self-check of the hash for the entire binary file and comparing it to a known-good value. This is inspired by message authentication codes and digital signatures.
- We also have a stretch goal of converting our binary to utilize self-encrypting/decrypting functionality that would utilize a shared secret to increase integrity and authenticity.

Requirement 3

The Attestation PIN and Replacement Token should be kept confidential.

These are the steps taken to address security requirement 3:

- Currently, the validate_pin and validate_token functions in application_processor.c use a fixed size buffer and perform no validation. We plan to add input sanitization to allow better size restrictions and to prevent non-number values.
- Enhance the attestation function to keep information confidential by replacing the StrComp function with constant time XOR or something similar.
- Use system time functions or similar functionality to measure the time taken for the input validation portion of the functions.
- We are aiming to implement similar functionality for the replacement token and replacement function for the attestation pair (as described under requirement 4). This would rely on self-encrypting functionality to protect the integrity of this function. We would encrypt the function and would be decrypted by generating the proper key using the digits from the replacement token. We currently have concerns about protecting against an attack where bad actors simply include their own replacement function that bypasses the need to decrypt ours. More investigations will need to be done as coding continues.

These are the steps that were not implemented in our final design:

- Our second goal is to prevent the exposure of the keys inside RAM during the supply chain attack. Currently we are thinking of attempting to interface with the MPU built into the arm processor to protect areas of memory. We are not sure if this will be efficient in a supply chain scenario. More testing is required.
    - If hardware mechanisms are not sufficient, we will rely on capturing the input from the host tool in a unique manner and processing it will further encryption values to prevent cleartext information from being leaked into RAM. Further details will be provided once we determine that the hardware is not sufficient.

Requirement 4

Component Attestation Data should be kept confidential.

These are the steps taken to address security requirement 4:

- The attestation data is encrypted with a key generated using numbers present in the pin. Currently we are unsure whether we will use the pin value by itself or concatenate it with a random number (This will depend on whether the length of the pin is sufficient to produce to produce a complex enough key).

Requirement 5

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

These are the steps taken to address security requirement 5:

- We plan to rewrite the secure_send and secure_receive functions with WolfSSL. WolfSSL is an embedded TLS library that will allow us to provide integrity and authenticity to the communication between devices.

Deployment

In the deployment process in our design, several files are used. In this section, the files used in this process will be outlined.

The keygen.c file in the deployment folder generates a random AES key using the WolfSSL library and prints it in a C array format. This file utilizes a random number generator to generate a block of bytes, which is the AES key. The AES key is printed out in a C array format, one byte at a time in hexadecimal format, separated by commas.

```
// Generate a random AES key
if (wc_RNG_GenerateBlock(&rng, key, sizeof(key)) != 0) {
    fprintf(stderr, "Failed to generate AES key.\n");
    wc_FreeRng(&rng);
    return 1;
}
```

The makefile is modified from the original insecure design by setting the compiler and linker flags that are necessary for the WolfSSL library. When the Makefile is ran with no arguments, it will compile keygen.c into an executable, run it, and direct its output into global_secrets.h.

There are other files in this directory called fun_secrets.h and semiFun_secrets.h, which are the subkey and initialization vector, respectively. These files are used in the attestation process.

Attestation

Figure 2 outlines our attestation process. This section will outline the various parts of the diagram, using labels 1, 2, and 3.

Starting with label 1, the `ectf_build_comp` command is called to get attestation data, and that data is fed into ectf_params.h. The attest data is "surgically removed" from ectf_params.h and given to encryptparams.c. WolfCrypt is then used to encrypt with AES256 and is then encoded into Base64 to be presented as a string. This string is then given back to ectf_params.h, which is then given to component.img in the encrypted and encoded form.

Label 2 uses the `ectf_build_depl` command to call a make file to call a C program to create pseudo-random number generator data (PRNG). This data is fed into fun_secrets.h and semiFun_secrets.h to generate a subkey and initialization vector (IV). The subkey is used in WolfCrypt in the previous paragraph and sanitizePins.c. The IV is used in WolfCrypt in the previous paragraph and in application_processor.img.

Label 3 uses the `ectf_build_ap` command to generate the AP_PIN that is "surgically removed" from ectf_params.h, XORed with the subkey, encoded with Base64, and given back to ectf_params.h. This XORed and encrypted data is given to application_processor.img.

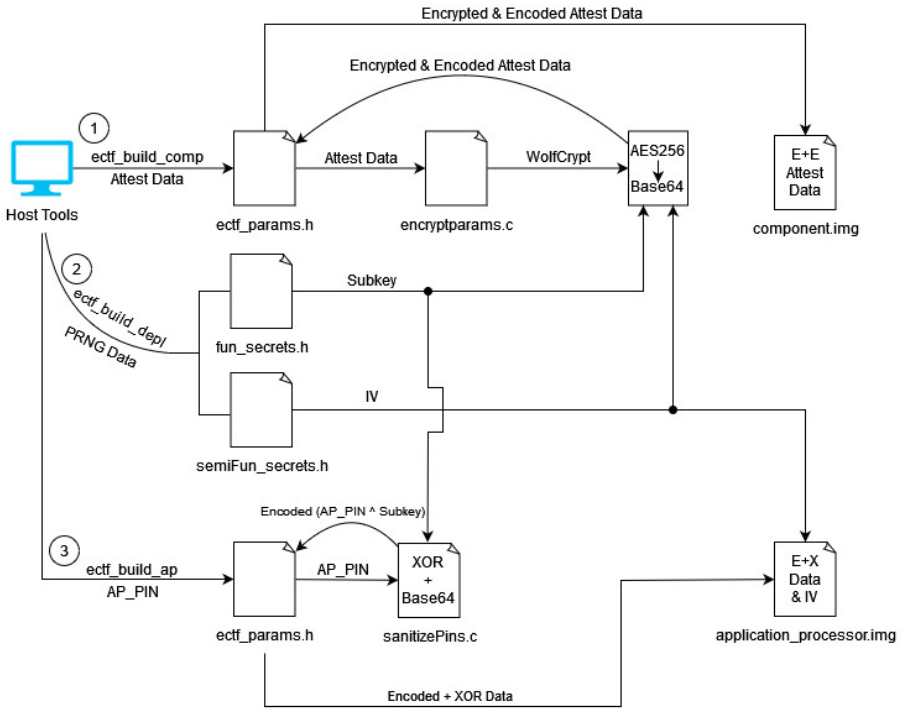

Figure 2: *Attestation process*

Replacement

In the replacement functionality of the medical device, we have implemented the SHA-256 cryptographic hash function. While using SHA-256 does not necessarily increase the level of security, it does offer a specific advantage. One of the primary benefits is that it effectively prevents the recovery of plain text pins. In other words, even if someone unauthorized gains access to the hashed pin, they would not be able to easily convert it back to its original plain text form. This adds an extra layer of security, making our system more robust against potential threats.

Secure Send/Receive

This section will describe our changes to the send and receive functions to enhance their security. For the send function, the inputs did not change. The function begins with creating a new buffer called the 'encrypted buffer'. It then verifies if the size of the message to be sent exceeds the maximum I2C message length. If it does, the function returns an error. If the message size is within the limit, we proceed to encrypt the message using the `encrypt_sym()` function from `simple_crypto.c`. This function accepts a buffer containing the plaintext, the length of the plaintext, the AES key, and the buffer where the ciphertext will be written. If this function does not return a value of 0, it indicates a failure in encrypting the message. Once we have the encrypted message, we calculate the padding. Finally, we use the `send_packet()` function to transmit the encrypted buffer along with the length of the padding.

The receive function commences with the initialization of an encrypted buffer. Before proceeding, we verify if there is any message to be received by utilizing the `poll_and_receive_packet()` function. If the length is less than 0, the function returns an error. If the length exceeds the size of the encrypted buffer, it indicates that the message is too large to be sent. If these checks are passed, we invoke the `decrypt_sym()` function. If the size of the decrypted buffer is less than 0, the function returns an error. We then copy the decrypted buffer into the buffer, making it ready for use. We calculate the length of the message and clear out the encrypted/decrypted buffer before exiting the function.

General Changes

There were many changes made throughout the insecure example to promote C best practices. Of these best practices, here are some of note:

- Properly Sized Buffers – There were several instances where the buffer allocated for some input text was larger than what the user was required to enter. To prevent malicious intent from the user, the buffers were adjusted to the correct size.
- Ending Buffers with Null Terminators – The buffers in the insecure example did not end with null terminators. It is important to end buffers in C with null terminators to ensure correct operation and prevent potential issues.
- Removed Buffer Overflows – We removed possible buffer overflows by means of input validation, using safe functions (i.e. using `fgets()` instead of `gets()`), and performing bounds checking.

Initial Designs Not Implemented

There were several challenges that we came across during the design phase that resulted in many of our initial goals remaining unimplemented. Firstly, we faced difficulties in implementing the challenge/response mechanism required for the first security measure as it was too expansive and many of the foundational items fell through. Specifically, the task of requesting a hash for a particular code section was unachievable due to the absence of a filesystem, preventing the hashing of arbitrary locations. This lack of file system also impeded our ability to decrypt arbitrary code, I believe this may still have been possible but further research was needed. Another missing function was the process of initiating a reboot, which was needed for many items, posed a significant obstacle as only the Power-on-Reset could be used. Finally, we just ran out of time/human resources to implement the timing functionality. Additionally, given more time, we would've liked to examine the watchdog functionality present in the MAX78000FHTR board. Notably, watchdogs allow for both fault tolerance and intrusion prevention and would be essential for any real-world medical device. However, we were concerned that without knowledge of the POST-BOOT procedures we would end up causing critical issues with the medical device.