Design Document - CyberAegis

The following document lists the security features that we implemented into our design.

Feature #1 - Buffer Overflow

One of the first things we noticed about the given design was the amount of C code used to write it. As C is a memory-intensive language, we immediately thought of a potential Buffer Overflow attack. These attacks can be quite dangerous to any program, especially those written in C because they can overflow memory storage, which overwrites memory data. Thus, it became crucial to try and find a way to prevent these attacks.

We started to look for places where the user types an input into the device, as this is where overflow attacks predominantly occur. After analyzing the code and getting a sense of the device, we found that the device is highly susceptible when the pin and tokens get entered in application_processor.c. Hence, we decided to add prevention mechanisms there.

Since we wanted to keep the code readable, we defined the maximum pin and token length at the top of the file. Another reason that we did this was to "maintain the competition scenario." We assumed that having the variables defined at the top would help an "administrator" have easy access to change the variable.

```c
#define MAX_PIN_LENGTH 49
#define MAX_TOKEN_LENGTH 49
```

Next, we gave buf, or the user input, a maximum byte capacity of the Maximum +1.

```c
int validate_pin() {
    char buf[MAX_PIN_LENGTH+1];
    // More code to follow
}
int validate_token() {
    char buf[MAX_TOKEN_LENGTH+1];
    // More code to follow
}
```

Finally, all we had to do was check if the length of buf was more than the previously defined Maximum. If the length was longer, we returned an error. We checked for length through the built-in strlen function. To check if the pin/token equals the input, we used the strcmp function. If the two match, then we return a Success. In all other cases, it returns an error in case of any unusual cases.

```c
int validate_pin() {
    char buf[MAX_PIN_LENGTH+1];
    recv_input("Enter pin: ", buf);
    if (strlen(buf) > MAX_PIN_LENGTH) {
        print_error("PIN too long!\n");
        return ERROR_RETURN;
    }
    if (!strcmp(buf, AP_PIN)) {
        print_debug("Pin Accepted!\n");
        return SUCCESS_RETURN;
    }
    print_error("Invalid PIN!\n");
    return ERROR_RETURN;
}
```

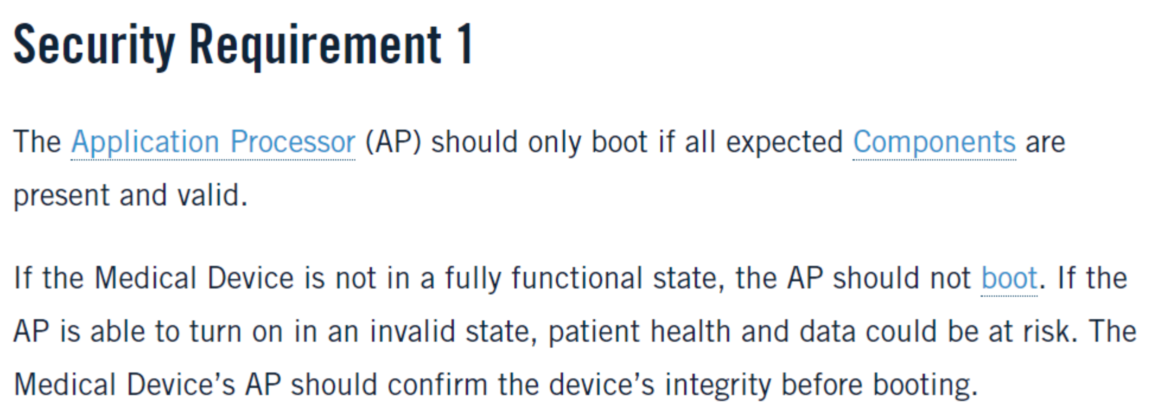
```c
int validate_token() {
    char buf[MAX_TOKEN_LENGTH+1];
    recv_input("Enter token: ", buf);
    if (strlen(buf) > MAX_TOKEN_LENGTH) {
        print_error("Token too long!\n");
        return ERROR_RETURN;
    }
    if (!strcmp(buf, AP_TOKEN)) {
        print_debug("Token Accepted!\n");
        return SUCCESS_RETURN;
    }
    print_error("Invalid Token!\n");
    return ERROR_RETURN;
}
```

In summary, to prevent Buffer Overflow attacks, we cross-checked the length of the inputs with the predefined values. If they were longer, we returned an error. If not, the code continued normally.

Feature #2 - Password validation

To ensure the confidentiality of SR1, a password was shared between the component and application processor to ensure that the components are valid.

To be specific, this security requirement stated:



Security Requirement 1

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

If the Medical Device is not in a fully functional state, the AP should not boot. If the AP is able to turn on in an invalid state, patient health and data could be at risk. The Medical Device's AP should confirm the device's integrity before booting.

Below is an outline of how this was implemented.

Addition of "password" array in application_processsor.c

- Here, we declare an array named 'password' with a type of 'uint_8'

```
typedef struct {
    uint32_t component_id;
    uint8_t password[13];
} validate_message;
```

Because the array has a size of 13 elements, it can hold a string of up to 12 characters plus a null terminator. This array stores the password received from the component during validation.

We did the same in component.c but used password[12] as the null terminator is not needed in component.c

The next thing that we defined in application_processor.c is:

```
char* passwordHash = PASS2;
```

Here, passwordHash is declared as a pointer to a character. It is assigned the value of PASS2, which is defined in our global_secrets.h file.

Similarly, we defined this in component.c

```
char* password = PASS2;
```

Here, password is once again declared as a pointer to a character. It is assigned the value of PASS2, which is defined in our global_secrets.h file. This is eventually used to ensure that password and passwordHash are the same values (there is no hashing involved, we just named one variable password and another passwordHash as a preliminary attempt to involve hashing, but unfortunately we were unable to complete it).

Now that our definitions are done, we must begin communication with the component and application processor. We did this through the premade functions, issue_cmd and send_packet_and_ack. That being said, we did have to make some modifications to the content of the transmit_buffer.

We did this in process_validate. It is here that when the AP requests a validation, the component responds with the component ID and password.

```
for (int i = 0; i < 12; i++) {
    transmit_buffer[4+i] = (uint8_t)password[i];
}
send_packet_and_ack(16U, transmit_buffer);
```

This loop iterates over the first 12 characters of the 'password' array. It then copies each character into the 'transmit_buffer.'

1. The loop counter starts at 0
2. The loop continues executing until 'i' is less than 12 (because we have 12 characters total)
3. The characters of the 'password' array are copied into 'transmit_buffer'
4. After each iteration, the loop counter increases by 1 (Purpose of i++)
5. Once i reaches 12, the condition 'i<12' becomes false, and the loop terminates
6. Upon termination, the transmit_buffer is sent with a length of 16 bytes (16U)

Finally, we added to the if condition for validation to include our password. Under validate_components:

```
validate->password[12] = '\0';
        // Check that the result is correct
if (validate->component_id != flash_status.component_ids[i] ||
strcmp((char*)validate->password, passwordHash)) {
    //print_info("a=%s\n", (char*)validate->password); (USED FOR DEBUGGING)
    //print_info("b=%s\n", passwordHash); (USED FOR DEBUGGING)
    return ERROR_RETURN;
}
}
    return SUCCESS_RETURN;
```

First, we terminate the string of the password array with the null terminator ‘\0’

Then, this if statement runs through two cases. It checks whether the component ID, received from validation, is **not** equal to the component ID stored in flash memory, and also compares the received password from the component (password) with the password we defined earlier in the application processor (passwordHash) using the strcmp function. If either condition is evaluated as true, then an error is thrown. In either case, the components have not been validated correctly, either through their ID or through the defined password. If both aspects match, the code would continue like normal.

Hence, we got the components to validate through their component ID AND password, meaning that both would have to match for the component to be considered “valid”.