MITRE eCTF Final Paper - Yellow Hackets

1st Henry Bui
Georgia Tech
Atlanta, U.S.
hbui43@gatech.edu

2nd Brianna Bumpus
University of North Georgia
Dahlonega, U.S.
bnbump3131@ung.edu

3rd Levi Doyle
Georgia Tech
Atlanta, U.S.
ldoyle9@gatech.edu

4th Andrew Graffeo
Georgia Tech
Atlanta, U.S.
agraffeo6@gatech.edu

5th Nashad Mohamed
Georgia Tech
Atlanta, U.S.
nmohamed9@gatech.edu

6th Mahta Tavafoghi
Georgia Tech
Atlanta, U.S.
mtavafoghi3@gatech.edu

7th Alan Zheng
Georgia Tech
Atlanta, U.S.
azheng74@gatech.edu

***Abstract*—This paper describes the MITRE eCTF 2024 competition and details as well as the Yellow Hackets team's implementation and security methods used to meet the various requirements of the competition.**

I. Introduction

The team's main tasks for the MITRE eCTF challenge are to construct a cryptographically secure infrastructure for an embedded system (specifically a medical device system) and to attack a system that has been secured by another team.

The medical system, which is referred to as the Medical Infrastructure Supply Chain (MISC) System, consists of 3 MAX78000FTHR boards welded onto a printed circuit board, one board acting as the Application Processor (AP) which communicates with the host computer, and two Component boards that represent the various components of medical equipment the system may be applied to.

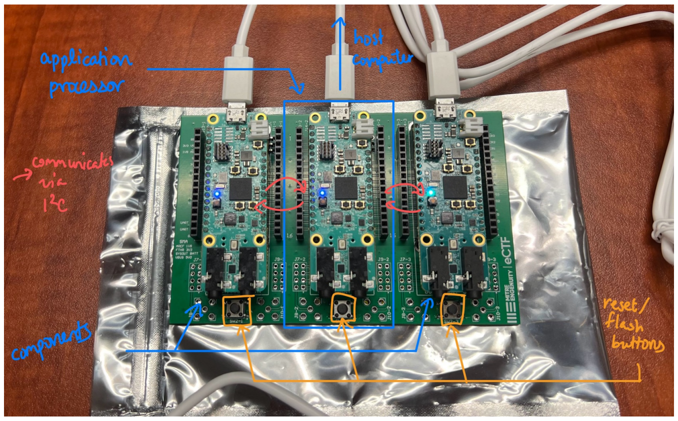


Fig. 1. This is the MISC board when it is fully set up. One board (here, the middle board) is labeled to be the application processor. It connects to the host computer via cable. It communicates to the component boards by I^2C. Finally, there are reset/flash buttons that can be used to load new firmware onto the board (in the updating process).

These systems can be flashed with the team's own firmware implementation. This process is known as "updating" the firmware. Though all competitors are given a reference firmware [1] that fulfills the functional requirements of the MISC system (i.e., the ability to boot, the ability for a component to be replaced, the ability for a component to be attested (queried for personal data), the ability for the application processor to list components, and the ability to send messages between the boards via I^2C), the team's task as the defenders is to secure the system so that attackers cannot take control of the system. In particular, the team has 5 security requirements to fulfill:

1) The AP should only boot when its components are all present and valid.
2) The component should only boot with a valid AP.
3) The attestation pin and replacement token (secrets that are needed to perform attestation and replacement respectively).
4) The attestation data (data obtained from a component via attestation) should be confidential.
5) The AP and components should be able to communicate with each other with integrity and authenticity.

The team can access these boards via the host computer (basically any system connected to the boards). Using the shell scripts they provided us (e.g., `ectf_update`, `ectf_attestation`, `ectf_replace`), which allows the team to command the devices and test the functional requirements.

The team was also tasked with attacking other competitors' designs. However, due to issues surrounding the completion of the team's design (particularly with security requirement 4), the team never reached this phase and unfortunately did not have the chance to attack other designs.

II. Protection and Security Details

In order to finish the security requirements efficiently, each security task was divided amongst the members of the team. For many of the goals, the team utilized the WolfSSL library [3] to write the implementations for many of the cryptographic procedures required to secure the system.

A. The Application Processor (AP) should only boot if all expected Components are present and valid. Components should only boot after being commanded to by a valid AP that has confirmed the integrity of the device.

Levi Doyle and Alan Zheng primarily worked on these security requirements.

1) Initial Implementation: To verify each component is valid, the deployment generates a SHA-256 hash for a part of the components' firmware and stores that data in the AP.

SHA-256 is a hashing algorithm that is very secure due to the fact that it is near impossible to reverse the input from the output. The algorithm takes the original message, a salt, and padding for security. MD5, another hashing algorithm, is not very secure since it's very easy to generate hash collisions. This makes tampering simple, which is why the team elected to use the SHA-256 instead of MD5 to hash the application processor and component files.

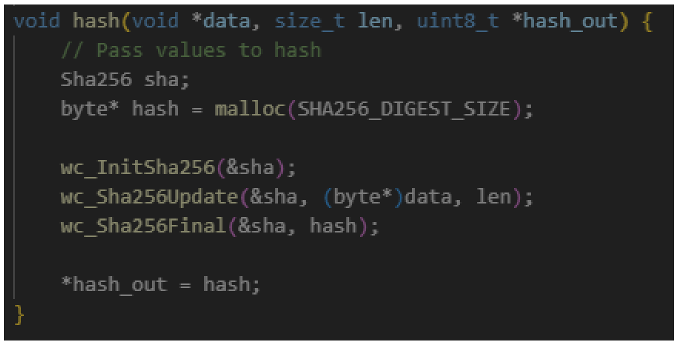
During boot, the AP recomputes the hash of the component's firmware, and check if it matches the AP's stored hash. If they match, the components are verified and the boot proceeds. If they do not match, the component firmware was tampered or compromised and the AP rejects the boot.

Similarly to how the AP validates the components, the components will also use a stored hash to validate the AP. A SHA-256 hash of the AP's firmware is generated and saved to a separate file, as simply inserting it into comparison code is impossible due to creating an infinite cycle of needing to update the AP and components' hashes due to each others hashing changing. This is done as part of the device's build.

During boot, the components utilize WolfSSL's SHA-256 functionality (See fig. 2) to create a hash of the AP's firmware and matches that to the stored valid hash expected by the component. If these hashes match, the components have verified that the AP is valid, and acknowledge the AP's request to boot. If these hashes do not match, the AP's firmware has possibly been tampered with or compromised, and the booting process ends, returning an error.

2) Flaws of Initial Implementation: While SHA-256 hash generation and checking against the valid hash was a correct approach, the initial implementation had a flawed design based on assumptions made about the functionality of the feather boards. The initial implementation assumed that the AP could simply access the components' firmware and then create a hash to check against the valid hash, and vice versa. This is not possible as each feather board only has access to its own firmware, and I^2C communication between the boards (which the initial implementation did not use at all) would not support the transfer of such a large amount of data. Additionally, the valid hash was stored in a local file, which was insecure and in an unreadable format for the firmware.

3) Final Implementation: The final implementation utilized the Simple Flash and Simple I^2C libraries provided with the default firmware to properly access the flash memory of the feather boards and send information between them using the I^2C board. Additionally, it stores the valid firmware hashes in the secrets file, that implements the valid hashes as secret variables that the firmware implementation will have access to. When the AP boots, it reads its own firmware from flash memory and creates an SHA-256 hash out of it. It then begins to verify each component, prompting them to create SHA-256 hashes of their firmwares and send them over I^2C to the AP. The AP will validate each hash by checking it with the valid component hash accessed from the secret variable. If the hash matches, the components can proceed to boot. The components will request the AP's generated hash over I^2C, and upon receiving it will check it against the valid AP hash accessed from the secret variable. If the hashes match, the component will proceed with its boot. If any hashes do not match, the verifying device will cancel its boot process and return an error.



```
void hash(void *data, size_t len, uint8_t *hash_out) {
    // Pass values to hash
    Sha256 sha;
    byte* hash = malloc(SHA256_DIGEST_SIZE);

    wc_InitSha256(&sha);
    wc_Sha256Update(&sha, (byte*)data, len);
    wc_Sha256Final(&sha, hash);

    *hash_out = hash;
}
```

Fig. 2. WolfSSL's SHA-256 functionality as used in the implementation. This function takes in a pointer to the data to be hashed and the length of the data, hashes it using SHA-256, and saves it to a specified point.

While a firmware implementation was complete for these requirements, time constraints and limited access to the testing hardware prevented the implementation to be tested and verified. However, the design is valid and fulfills the security requirements.

B. The Attestation PIN and Replacement Token should be kept confidential.

This goal was mainly tackled by Brianna Bumpus.

In the reference design, the firmware authorizes the attestation and replacement operations by *directly string-comparing* the user's password input against the firmware's attestation PIN and replacement token.

To replace this insecure verification of string-comparison, the team instead opted to store and compare against a SHA-256 hash of the PIN and token in the firmware. The team also later added salting to the authorization process to further secure these tokens.

1) Hashing: When the application processor needs to attest or replace a component, it takes the pin or token passed by the user and applies a SHA-256 hash against this input. Then, this can be compared to the stored SHA-256 hash of the PIN and token to verify the user passwords.

As the PIN and token are not stored as plaintext within the firmware, it would not be possible for others to reverse-engineer the binary to extract the PIN and token used.

The MITRE organizers provided a WolfSSL cryptography example using a MD5 hashing algorithm, but upon review the team determined the MD5 hash to be insufficient. The WolfSSL library contains multiple hashing algorithms such as, MD2/4/5, SHA 128/224/256/384/512, RIPEMD, BLAKE2, etc. [3] After reviewing the options, using a SHA256 hash algorithm seemed to be the most holistic, and fit the team's needs best.

When compared to SHA256, MD5 is considerably less resistant to attacks. The largest reason for this is due to the difference in the size of the hashed outputs; MD5 hashes are 128-bits, while SHA256 is a larger 256-bits hash. By having a larger size, the hash becomes more resistant to collision attacks, where an attacker attempts to find two inputs that produce the same output after being hashed. Furthermore, there are efficient collision attacks available that have been specifically created for targeting MD5 algorithms and known vulnerabilities.

Upon deciding on SHA256 as the hash algorithm, the first step for implementation of the security requirement was to create a new hashing function that would leverage the WolfSSL library. To do this, the team utilized `wc_InitSha256()` to initialize the SHA256 structure, next `wc_Sha256Update()` to hash a byte array, and then `wc_Sha256Final()` to finalize the data hashing and place the hash into a byte array. [3]

After the new hashing function was created, it was applied within in the `application_proccessor.c` code. The AP has two functions for validating an attestation pin and replacement token, respectively. Within each function a string comparison is done using `!strcmp()`, which accepts two strings and compares them to determine if they are a match.

Since the design utilizes string comparison, the hash function would need updated to output a string (null-terminated char array) as opposed to a byte array. A for-loop was added to the function as a solution to convert the byte array to a hexadecimal string, to allow for comparison via `!strcmp()`.

The next step after the creation and update of the hash function was to use it to secure and keep the Attestation PIN and Replacement Token confidential. The initial plan for the implementation was to hash the Pin and Token in the python build AP file, so that the plaintext pin and token are not anywhere in the `application_proccessor.c` file. By keeping the plaintext separate, it prevents the Pin or Token from being accessible by reverse engineering the binary.

2) Hashing + Salting: Salting adds an additional layer of security over hashing for password-type verification by personalizing the hash to each session.

Since the authorization happens wholly within the application processor, the addition of salting only needed to be done within the application processor without any necessary coordination to the components.

To implement salting onto the application processor, the team computed a random block of 8 bytes via the built-in true-random number generator of the MAX78000FTHR board every time the system booted. Once a salt is computed, the salt block is XORed with the plaintext attestation pin and replacement tokens before SHA-256 hashed.

Then, during authorization of attestation and replacement, the same process is applied to the plaintext (the salt block is XORed with the plaintext before SHA-256 hashed) and then compared against the salted hash of the firmware's built-in attestation pin and replacement token. The general process is described with fig. 3.

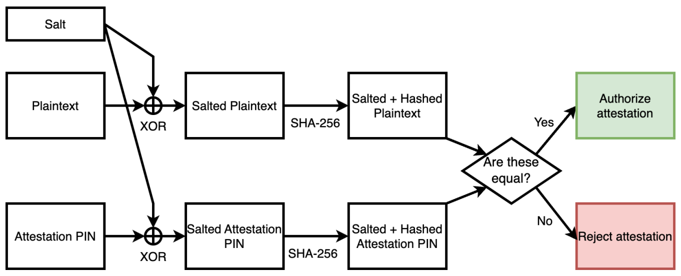


Fig. 3. Diagram of the attestation authorization process. A similar process is done with the replacement authorization process.

C. Component Attestation Data should be kept confidential.

This goal was mainly tackled by Nashad Mohamed and Mahta Tavafoghi.

In the process of attestation, the component sends its attestation data (i.e., its customer name, its manufacturing location, and its date) to the application processor. By default (in the insecure design), this is sent in plaintext through the I^2C channel from the components to the application processor.

The team went through two designs to implement this goal: an AES-XTS block cipher implementation and an RSA implementation.

1) AES-XTS Block Cipher Implementation: The original plan was to use AES-XTS block cipher in order to encrypt and decrypt the attestation data. Advanced Encryption Standard (AES) is a symmetric block cipher widely used by the U.S government to encrypt sensitive data. Data is split into blocks of 128 bits and the same key is utilized to encrypt and decrypt data. XTS is one of the AES block cipher modes which is more secure, as it eliminates some side channel attacks and potential vulnerabilities. XTS utilizes two AES keys with one key for AES block encryption and another to encrypt a "tweak key". Every data unit is given a tweak value that is non-negative, and when encrypted, it is converted into a little-endian byte array. XTS achieves more security as the plaintext is basically double-encrypted with the use of two independent keys.

The AES-XTS approach seemed too complicated to tackle, and the team opted to move towards an RSA approach, as the asymmetric key setup in RSA seemed very fitting to the attestation task and the team was more familiar with how RSA functioned.

2) RSA Implementation: Since the components only need to transmit attestation data and the AP only needs to receive attestation data, the team opted to encrypt and decrypt the data using RSA. RSA (Rivest-Shamir-Adleman) is a type of

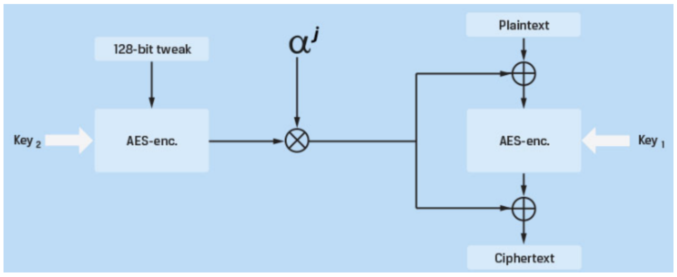


Fig. 4. Diagram of AES-XTS block encryption procedure

encryption that enables a public key and is usually used to protect sensitive data. It is usually used for data that is being sent through an insecure network. In RSA encryption there is a private and public key. It is widely used since it assures confidentiality and that no party can deny that it sent/received a message through encryption. In the design, a public-private key pair is generated during deployment. The component can use the public key in order to encrypt a message and send it to the application processor who can use the private key in order to decrypt the message. This helps protect the MISC system against attackers as the encrypted data will be useless without knowing the private key.

While the team did successfully implement RSA during the attestation process, the team also failed to get the implementation to pass through tests. Several unexpected issues were faced in the process of attempting to pass implementation tests. Notably, this included:

1) Size issues with the RSA keys
2) Timing issues with the RSA encryption process

Regarding problem 1, the team had identified that 1024-bit private RSA keys were not large enough to encrypt the largest possible attestation message (attestation messages could be at maximum 206 bytes, but a 1024-bit private RSA key could only encrypt 117 byte messages). The simple solution would have been to upgrade to 2048-bit private RSA keys (which could encrypt 245-byte messages); however, that caused a second issue (problem 2).

Regarding problem 2, the attestation had to complete in 3 seconds (this was required by the rules of competition). Notably, encrypting with a 2048-bit key took approximately 7 seconds (i.e., too much time). The solution to this problem was to break up the message and encrypt the parts separately with 1024-bit private keys. This did work locally (it took approximately 2 seconds total), but on the global tests provided by the maintainers, the tests actually took 3+ seconds, which thus made the solution incomplete for the purposes of the competition.

In the future, if the team were to remedy this issue, they could try to downscale by using 512-bit private RSA keys **or** dropping RSA entirely and using a simpler asymmetric-key cryptosystem.

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

This goal was mainly tackled by Henry Bui and Andrew Graffeo.

The team attempted three approaches to the secure the integrity and authenticity of messages through the post-boot channels:

1) Secure Sockets Layer (SSL): The team first attempted to implement SSL through the channels. SSL is intended for networks, and relies on several handshakes between the source and destination parties. The team soon realized that the size of the handshake packets and the number of handshake packets would be too much for the I^2C channel between the AP and components. Additionally, the implementation of SSL required the addition of the SSL portion of the WolfSSL library which doubled the size of the firmware beyond the competition's firmware limit (of 226KB).

Thus, we switched to a simpler communication protocol that was better fit for embedded systems—ECIES.

2) Elliptic Curve Integrated Encryption Scheme (ECIES): The second approach the team took is to implement the Elliptic Curve Integrated Encryption Scheme (ECIES) to create keys to encode and decode messages between the application processor and components.

ECIES is a system to enable cryptographically secure communication between two parties (a client and a server), similar to what SSL does [2]. ECIES creates an asymmetric public-private key pair between the server (which will be the application processor) and the client (which will be the components).

The two parties perform a handshake where they send each other their public keys. This creates a situation where the client has its own private key and the server's public key, and the server has its own private key and the client's public key. Together, these represent the "shared secret."

This shared secret isn't used directly to encrypt messages, but rather, every time the two parties wish to communicate, they produce an encryption salt. To perform communication, the following process occurs (which is depicted in Fig. 5):

1) The transmitter sends their salt over.
2) The receiver sends their salt over.
3) The transmitter, using the shared secret and salt, encrypt plain text and send the encrypted text over.
4) The receiver, using their shared secret and salt they have, decrypt the encrypted text.

The team also decided to take a step back from the ECIES implementation for similar reasons as the departure of the SSL implementation. The handshakes for ECIES, while much simpler to implement to SSL (and with more direct examples to base the implementation on [2]), still was complicated to implement. Additionally, the team also had around a month remaining before the Attack Phase ended, so the implementation was further downscaled to make the implementation more doable to complete before the end of the Attack Phase.

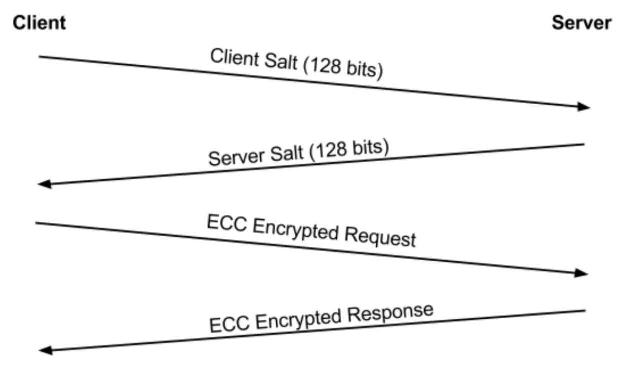


Fig. 5. The communication path through ECIES. Here, the client acts as a transmitter and the server acts as the receiver.

3) Advanced Encryption Standard, Cipher Block Chaining Mode (AES-CBC): For the third version of a solution to security requirement 5, the team decided to take a simple symmetric encryption approach. The team figured that a symmetric key approach was sufficient for secure sending/receiving between the application processor and components as that would reduce the number of keys that had to be generated between the application processor and components (For a symmetric scheme, only 1 key would need to be produced between the application processor and components. In contrast, for an asymmetric scheme, at least 4 keys would be needed—a public and private key for messages sent from the application processor, and a public and private key for messages sent from the components).

In the insecure design, the Advanced Encryption Standard (AES) was used to implement a symmetric encryption/decryption scheme (a scheme that uses the same key to encrypt and decrypt a given message). However, it used the Electronic Codebook (ECB) mode of AES, which is the simplest and least secure implementation of AES. In AES-ECB, the plaintext is split into "blocks", each of which undergo the same AES encryption process with the same key. AES-ECB is known to be cryptographically insecure because of its failure to hide patterns from the plaintext (the clearest example of this is Wikipedia's Tux example, which can be seen in fig. 6).

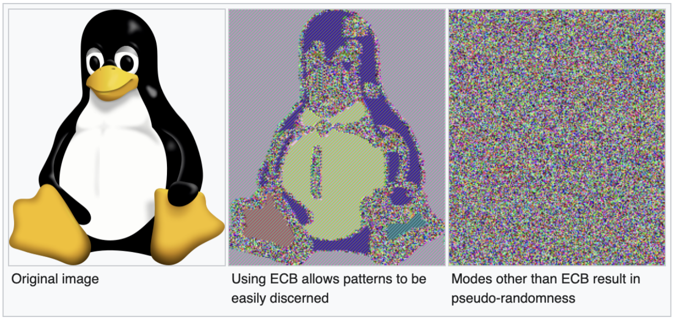

Original image
Using ECB allows patterns to be easily discerned
Modes other than ECB result in pseudo-randomness

Fig. 6. Issues regarding the use of AES-ECB.

To curtail this, we decided to move one step up and use the Cipher Block Chaining mode of AES (AES-CBC). Like AES-ECB, the plaintext is split into "blocks", which are encrypted separately. However, unlike AES-ECB, after each encryption, the ciphertext block is XOR'd with the next plaintext block before the AES encryption occurs. This removes the patterns seen with AES-ECB as the plaintext data is essentially scrambled before being encrypted. (See the differences between AES-ECB and AES-CBC's implementation in fig. 7).

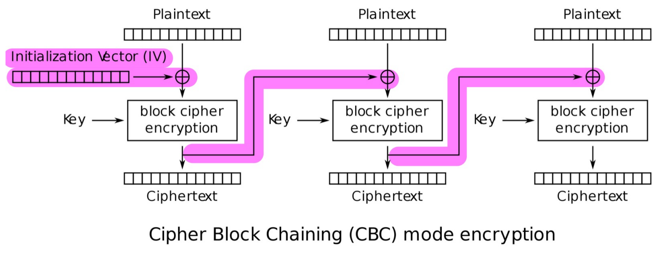


Fig. 7. This chart describes how AES-CBC functions. Highlighted in pink are the additions AES-CBC when compared to AES-ECB.

The implementation of AES-CBC was done relatively simply. The process included the following steps:

1) Create both an AES key and an AES initialization vector.
2) Replace the `encrypt_sym` and `decrypt_sym` functions implemented in `simple_crypto.c` by the insecure design. This involved replacing calls of `wc_AesEncryptDirect` and `wc_AesDecryptDirect` (AES-ECB) in these functions to `wc_AesCbcEncrypt` and `wc_AesCbcDecrypt` (AES-CBC).
3) Use `encrypt_sym` and `decrypt_sym` to implement the `secure_send` and `secure_receive` methods that had to be secured for security requirement 5.

III. Vulnerability Analysis and Attack Details

As mentioned, due to issues regarding the implementation of certain security requirements (particularly #4), the team never managed to get to the Handoff and Attack Phases when attacking of other competing teams' boards would have been attacked and reverse-engineered. Reverse engineering is helpful for understanding the inner workings of the medical device and identifying any security vulnerabilities.

However, the team did do some research into reverse-engineering and attacking. One tool that can be used for reverse engineering is Ghidra, which is an open source software developed by the National Security Agency (NSA). Ghidra allows a user to disassemble code and step through forward and backward to understand how the functions are mapped out. It is commonly used to investigate malware. A user imports a binary file that is then decompiled. Ghidra's window has a ”Symbol Tree” section that displays import, export, function, labels, classes, and namespaces of a binary file. This is helpful when trying to understand how the code is organized and which functions to look in.

IV. Conclusion

The team learned a lot about embedded systems, cryptography, and defensive cybersecurity during the process of designing firmware for the MAX78000FTHR.

In terms of cryptography, the team learned about the differences between different hashing algorithms (MD5 and SHA-256), salting, AES-XTS, AES-CBC, RSA, and communication protocols (SSL and ECIES). Even when the team was not able to implement all of the features in full, researching and *attempting* to implement each feature provided knowledge about the internal workings of each system.

In terms of embedded systems, the team learned that embedded systems' computational and storage limits are significantly smaller than the laptops and devices everyone has, and as a result, designing a secure, protected system has to take those computational and storage limits into account.

The team had faced several issues regarding the timely implementation of each of the security requirements. While it was a good idea to task separate team members to separate requirements, the team still faced issues between team members lacking sufficient knowledge to complete the security requirement or the original design being too complex or computationally expensive to perform on the embedded system. In the future, designing the process should fall under these priorities, so that the design can be completed in a timely pace:

1) Devise the simplest idea that *works*.
2) Devise any variants to the idea that would help secure the system more.
3) Devise any new ideas that are more secure, more complex, and may demand more from the embedded system.

This is the approach that was taken for security requirement 3, which was the only requirement that was truly completed in full. Applying these priorities to the remaining requirements would likely have helped the team complete before the start of the attack phase.

References

[1] MITRE Corporation, "eCTF Insecure Example," GitHub. [Online]. Available: https://github.com/mitre-cyber-academy/2024-ectf-insecure-example

[2] T. Ouska, "BTLE Secure Message Exchange." wolfSSL, Nov. 01, 2013. [Online]. Available: https://github.com/wolfSSL/wolfssl-examples/blob/master/btle/ecies/BTLESecureMessageExchange.pdf

[3] wolfSSL, "wolfSSL Manual." [Online]. Available: https://www.wolfssl.com/documentation/manuals/wolfssl/index.html

[4] ISSP, "Reverse Engineering with Ghidra," ISSP Global, Apr. 23, 2019. https://www.issp.com/post/reverse-engineering-with-ghidra