HackieBird

Virginia Tech

Raghav Agrawal, Kyla Coles, Wesley Flynn, Erin Freck, Christian Johnson, Jack Kolenbrander, Thomas Rydzewski, Raj Sahu
Advised by: Dr. David Raymond PhD and Dr. Matthew Hicks PhD
April 24, 2023

Design Overview

The following architecture was implemented to secure the functional requirements of the competition:

1. **UNLOCK CAR:** A secret password shared between Car and Fob is used to perform Challenge Handshake Authentication Protocol. Since the attack binaries are encrypted, the password is stored as a macro in the code.
2. **PAIR FOB:** The pairing pin is hashed and stored on the paired fob, which thwarts the attack in case of fob memory leakage.
3. **ENABLE FEATURE:** ECDSA is used to sign packaged features and verify them on the fob during feature enable request.

The car secrets and feature secrets are stored on the EEPROM, which is locked with a password stored as a macro in the code.

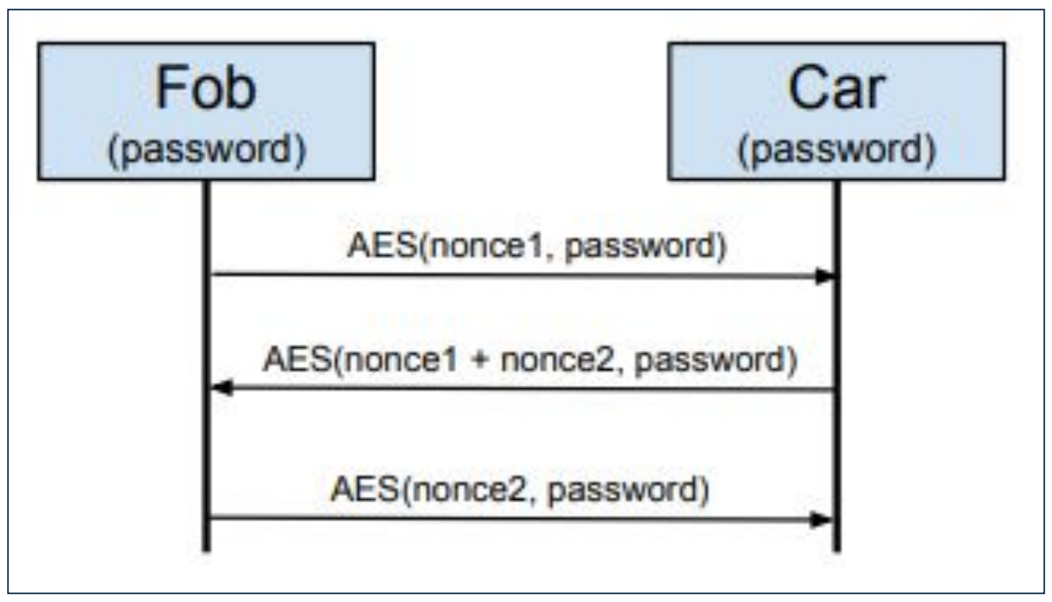
Defensive Highlight

Included Security Measures:

- **AES, SHA256, ECC -** Utilized in all parts of the design to encrypt information in transit and at rest.
- **Nonce -** Utilized during pairing and unlocking to prevent replay attacks.
- **Digital Signatures -** Utilized for packaging and enabling a feature to verify a device's identity.

Challenge Handshake Protocol

The challenge handshake protocol was implemented for unlocking the car. Using a predetermined, pre-shared key, the fob initiates unlocking by encrypting a nonce with AES and passing it to the car. The car then decrypts the nonce, appends its own, encrypts the concatenation, and sends it back to the fob. The fob then decrypts the nonces, checks if it got its original one back, then encrypts and sends the second one back to the car. The car then verifies the nonce and unlocks if they match.



Utilizing this type of challenge handshake was the only way we could find that would be secure against a replay attack or unknown entity without using a third party. However, other teams broke it so the implementation must not have been robust. We are still in the process of analyzing why this method did not work, but currently our best guess is from either poorly managed memory, not using a good enough RNG, or not securing the key well enough.

We really struggled on a proper RNG method, and with more time, we would have liked to further explore NIST SP 800-90A. As we were already behind in getting to the attack phase, we decided to throw together something a bit haphazardly.

Offensive Highlight

Attempted Attacks:

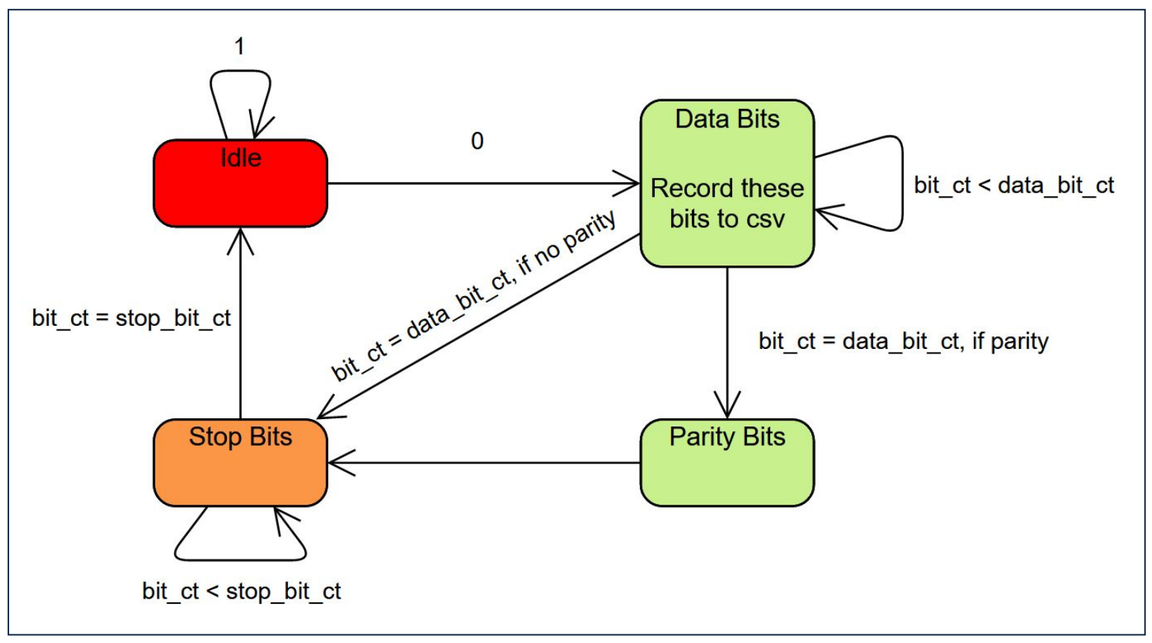
The first step we took towards attacking was the parsing of the given UART csv. We created a program in python the prompts the user for the input file name, baud rate, number of data bits, parity bit, and number of stop bits. All of these values (except the csv filename) can be found in each team's uart.c file.

The program has two parts. The first fills in the bit that were lost in the csv file's recording. It does this by comparing timestamps and inserting the appropriate number of bits into a pandas dataframe. The diagram below describes this procedure.

Timestamp	Bit
0.002	0
0.0034	1

Insert top_bit[(time_top – time_bottom)baud_rate – 1]

The second part of the program is a finite state machine that takes in each of the bit from the expanded list as an input and outputs the raw data contained in the UART transmissions. The data and stop bit states are repeated the appropriate number of times based on the input parameters at the beginning of the program. This is what the bit_ct variables are in the FSM diagram below.



We used the output from this program to attempt, but without success, replay and brute force attacks. If we had more time, we would specifically craft the data sent over based on the byte sequence found by our program to exploit the system.

References

1. Gladman, Brian (2023) Sha [Source Code] https://github.com/BrianGladman/sha
2. MacKay, Ken (2023) Micro-ecc [Source code] https://github.com/kmackay/micro-ecc