2023 MITRE eCTF Design Documentation and Attack Approach

1st Dinko Dermendzhiev
Georgia Tech
Atlanta, U.S.
dinko.dermendzhiev@gatech.edu

2nd Katherine Paton-Smith
Georgia Tech
Atlanta, U.S.
kpatonsmith@gatech.edu

3rd Adith Devakonda
Georgia Tech
Atlanta, U.S.
adevakonda3@gatech.edu

4th Levi Doyle
Georgia Tech
Atlanta, U.S.
ldoyle9@gatech.edu

5th Lindsay Estrella
Georgia Tech
Atlanta, U.S.
lestrella7@gatech.edu

6th Veena Gonugondla
Georgia Tech
Atlanta, U.S.
vgonugondla3@gatech.edu

7th Ritvik Verma
Georgia Tech
Atlanta, U.S.
rverma83@gatech.edu

8th John Zhang
Georgia Tech
Atlanta, U.S.
jzhang3213@gatech.edu

***Abstract*—The 2023 MITRE Embedded CTF competition tasks teams with developing two firmware builds: car and fob. This document outlines the design decisions and implementation of this firmware as it relates to a few important security measures. In anticipation of the attack phase part of the competition, Section III details the work completed thus far by the reverse engineering sub-team.**

I. Build Environment

The following software will be utilized on each machine as part of the eCTF: Git, Docker, Python, Stellaris ICDI Drivers, and UNIFLASH. Git, the open-source version control system, allows for collaboration across the team on a single code base while maintaining a history of edits. Docker will create the environment in which the host tools execute by packaging all of the tools with the required software. As it is used in the provided eCTF tool repositories, the Python programming language will be used for its powerful development capability. The Texas Instruments TM4C123G LaunchPad Evaluation Kit is the development board for this competition. It contains an integrated In-Circuit Debug Interface (ICDI) to support the programming and debugging of the hardware. As such, the appropriate Stellaris ICDI Drivers are required. UNIFLASH will be used to program the flash memory of the development board.

II. Functional Requirements

There are four tools to the design: enable, package, unlock, and pair. They meet the functional requirements by implementing: sending a packaged feature to a fob, creating a packaged feature, listening for unlock messages from the car while unlocking via button, and pairing an unpaired fob through a paired fob, respectively.

III. Design Phase: Security Measures

The following section describes the security goals and how each will be met in the design.

A. A car should only unlock and start when the user has an authentic fob that is paired with the car

Car and paired fob communications are encrypted using the Advanced Encryption Standard (AES) in CTR mode. This encryption protocol was chosen for both its relatively cheap cost of computation and its security properties across repeated messages.

An infamous difficulty, and potential vulnerability, with symmetric key encryption is the process of key exchange before any encryption ever occurs. If the protocol is compromised at this step, via interception of the secret key by a third party, any further encryption is insecure. To mitigate this in the firmware design, the AES key is created and exchanged at compile-time, thus the key is never transmitted in the open.

At a technical level, this is achieved via a Python script in the 'car' directory that generates a key, crafts a header file for use by the firmware, and exchanges the key through a shared JSON file. The script is run prior to compilation and utilizes the "secrets" Python library to instantiate the AES key. Then, a similar Python script in the 'fob' directory can access the JSON file to create its own header file. The respective header files then get compiled with each device build, and they are accessible via direct reference in each firmware file.

Due to the nature of fob and car interactions, whereby the same unlock message needs to be produced and transmitted time and time again, CTR mode was chosen to ensure security against replay attacks. In short, CTR (counter) mode generates a new initialization vector for every piece of data that is encrypted and sent. This means that the probability of two ciphertexts (encrypted messages) being identical is extremely low, despite conveying the same plain-text (decrypted) messages. To aid in the implementation of AES encryption, the 'tiny-AES-c' library by GitHub user 'kokke' is included. It provides the functions for standard encryption and decryption.

In the fob's firmware, upon boot-up, the private key and the unlock password are retrieved from secrets.h and the rand() function is seeded with a pseudo-random value (SysTick) in preparation of generating random IVs. Then, at each invocation of the unlock() method, the fob generates a new IV, computes the cipher-text, concatenates the two together, and sends this encrypted message to the car via the established UART connection. It is vital that the IV be communicated to the car as the decryption process is simply a reflection of the encryption process. It should be noted that it is safe to send the IV over plain text because 1) it will never be used again and 2) decryption without the key and under the assumption of reason 1 is virtually impossible.

In the car's firmware, once it receives a correct unlock signal, it proceeds to parse out the IV and ciphertext from the message buffer. It calls the exact same tiny-AES-c functions to perform CTR decryption on the cipher-text. Finally, it compares the input messages to it own copy, which was shared through the same Python generation scripts referenced earlier. If the sequence was successful, the car writes the last 64 bytes of EEPROM to UART, which consists of data regarding the enable features, and it calls the startCar method.

Looking forward, the only possible vulnerability in this scenario that the team anticipates is the acquisition of the secret key via memory forensics on the board's EEPROM. Methods and tools for packing and data obfuscation are being explored.

B. Revoking an attacker's physical access to a fob should also revoke their ability to unlock the associated car

The design provided in sub-section A covers this security requirement. Without access to the fob, an attacker would not have access to the private AES key and thus will not be able to accurately encrypt the unlock message. Furthermore, the attacker would have to know the mode of encryption and agreed-upon protocol for cipher-text formatting.

C. Observing the communications between a fob and a car while unlocking should not allow an attacker to unlock the car in the future

Previous unlock messages will not be able to be used in the future due to the use of AES in CTR mode symmetric encryption. As explained in sub-section A, this is because initialization vectors are never reused. In other words, old messages or communication between the car and the fob are rendered useless and out of date. This synchronous counter will make it hard for an attacker to recover any information or patterns.

D. Having an unpaired fob should not allow an attacker to unlock a car without a corresponding paired fob and pairing PIN

Without the paired fob and its pairing PIN generated previously, an attacker will not have the private AES key to sign messages. In addition, the PIN will not be stored in plain text but rather encrypted for added security.

E. A car owner should not be able to add new features to a fob that did not get packaged by the manufacturer

At the time of build, a secret key between the host and the fob is created and shared with the fob. The host then uses this key to encrypt any feature packages sent to the fob. The fob can then decrypt this installation file and authenticate that it's from the manufacturer. Without the correct installation file, the fob will not allow any modifications to be made as it can tell that it was not made by the manufacturer.

Modeled after the gen_secrets.py files for the car and fob, a gen_secrets.py file for the host was created to randomly generate an AES key and save it to the secrets file. The host Makefile was edited to run the gen_secrets.py script. The host key was added to the secrets of the fob in the fob's gen_secrets.py file.

During testing, there were issues where the AES python Library was outdated. Research indicated Pycryptodome is a replacement library. When testing with the new library, there was an issue where that the library was not recognized as installed. To fix this, the line RUN pip install pycryptodome was added into the docker file.

Initially, EAX mode was used in enable_tool for the AES key. The tiny-AES-c library implemented can encrypt and decrypt in C but when a feature is enabled it is sent to the fob in a python file. In other words, a python library was used to encrypt the message from the host, but the code to receive messages from host to the fob is in C. AES in the PyCrypto library and tiny-AES-c work the same, but tiny-AES-c does not have an EAX mode. So instead CTR mode is used for AES encryption to match the mode used in decryption. Changes include initializing a cipher and iv to implement CTR mode.

To send the encrypted feature package with the IV between the host and the fob, the struct python library was used. Its .pack() method was used to pack the encrypted feature package and IV together into one message to send. Then in the fob's firmware, structs were created to access the encrypted feature package and IV from the received message.

When testing with the boards, there were issues that the host could not access the secret file with the shared AES key. In the Docker environment, there are no build steps where the secrete volume used to store the AES key and tools volume where the host tools are run out of are both mounted in the same step. The Docker build environment was not allowed to be modified for the competition. So in the fob and host tools have no shared volume between them in which to share secrets. To fix this, the AES key is hard-coded and then hashed for the host and the fob. This is not a secure solution, but a functional one.

F. Access to a feature packaged for one car should not allow an attacker to enable the same feature on another car

The Car ID is sent in a feature package message. The fob then checks the Car ID before enabling the feature.

IV. Attack Phase

In the Mitre eCTF, teams are tasked with analyzing and attacking each other's designs in order to gain points. During the competition, teams are provided with access to various resources to find vulnerabilities, including source code, system documentation, and host tools. The following section describes the flags and approach of the attack sub-team for the attack phase.

A. Flags

There are six flags that can be obtained by compromising one or more security requirements: new car unlock, temporary fob access, passive unlock, leaked pairing pin, pin extract, and enable feature. Each flag's point value diminishes as more teams capture it, encouraging teams to attempt the difficult flags.

B. Tools

One of the tools that teams can utilize in the competition is GHIDRA, a reverse engineering framework released by the NSA. GHIDRA allows teams to analyze binaries, including executables and libraries, to understand their functionality and identify potential weaknesses. GHIDRA lifts from assembly to p-code to represent any of the different ISAs that exist on the market. As of GHIDRA 10.2, it can emulate in p-code so that firmware can be run in GHIDRA. x86 and ARM are compatible, however other architectures such as VLIW do not work well with GHIDRA because it becomes difficult to determine what code does statically.

The SVD-Loader script is a supplementary tool that builds parts of the memory map corresponding to the peripherals and registers based on a system view description file before analysis to accelerate the reverse engineering process. The python script at the root of the SVD-loader repository must be added to GHIDRA's script manager.

C. Setup

Binaries and other files are imported into a new GHIDRA project. The binary files for the car and fob firmware are in ARM Cortex 32-bit Little Endian and must be specified as such upon initial opening of the binaries. The Texas Instruments Tiva boards begin instruction execution at address x8000, which can be specified when first opening the binary to line up symbols better. However, in practice, some functions and labels before x8000 are necessary to fully map binaries to the source code. The SVD-loader script must be run before auto analysis of the binary. Opening the script manager in the code browser and running the SVD-loader script with the TM4C123GH6PM.svd sets up parts of the memory map before analysis. Running auto analysis with default settings after the script decompiles the binary and completes preparations for reverse analysis.

Additionally, elf files compiled with debug symbols can be analyzed to accelerate the process by providing labels to functions.

D. Attack Process

The first step of reverse engineering is to map out functions by observing function signatures, defined strings, function call trees, and the structure of function bodies and mapping them to functions in the source code. This step is greatly accelerated by usage of debug symbols and elf files. Next, the functions corresponding to inputs must be identified and analyzed. In the case of the fob, these appear to be UARTchar-get, UARTreadline, receiveBoardMessageByType, receiveAck, receiveBoardMessage, and GPIOpinRead. Tracing the function call trees from these functions and observing the assembly instructions and input methods of the functions gives rise to potential attacks using malformed inputs.