Attack Phase Group Assignment

1. Introduction

The Embedded Capture the Flag (eCTF) is an embedded security competition run over the spring semester by MITRE Engenuity that puts participants through the experience of trying to create a secure system and then learning from their mistakes. Our pivotal task centered on programming firmware for the MAX78000FTHR board. Using Nix for consistent build environments helped us standardize and clarify our developmental processes. Additionally, the competition encouraged using C and other programming languages for our solutions including Python, C/C++, and Cortex-Debug. This approach allowed us to tackle the challenge innovatively within a set framework.

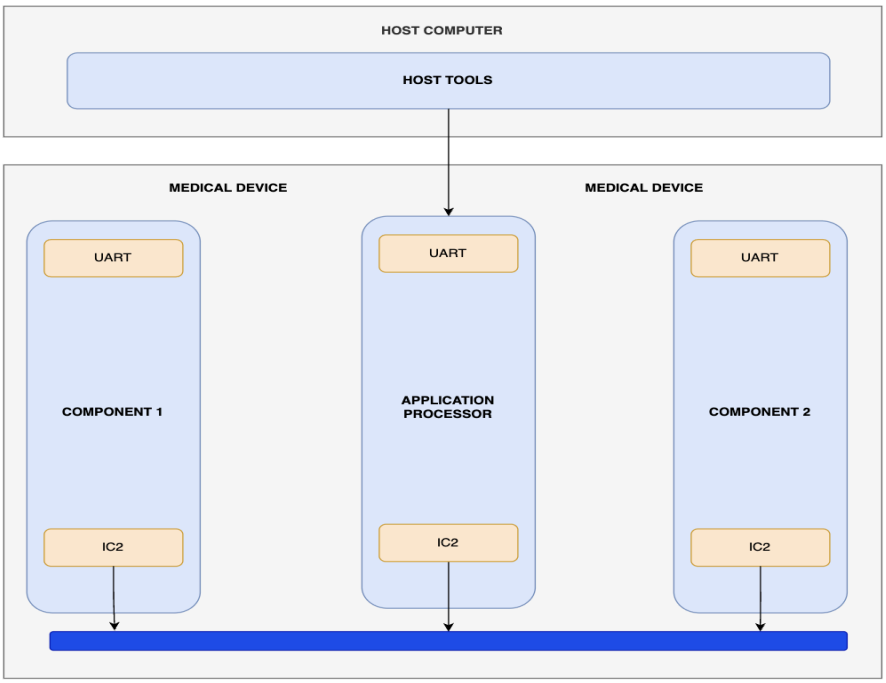

Figure 1 *High-level overview of the system architecture*

The Medical Device (MISC) and the Host Computer communicate over a serial interface using various Host Tools on the Host Computer, which is a general-purpose computer such as your laptop or desktop. These tools are used to read back status messages and data, as well as to trigger the medical device's numerous functionalities.

Attack Phase Group Assignment

The Medical Device, or MISC, is the primary focus of our team's effort. An Application Processor (AP) and two Components linked by a common I2C bus make up a medical device. The custom PCB will connect each Analog Device MAX78000FTHR development board when implemented. It should be fine with what the end gadget is used for because MISC is utilized to cover a variety of medical devices (such as infusion pumps).

The medical device's "brains" are called the Application Processor (AP). It manages all communications with the host computer, plans the MISC functionality needed to implement, and performs all of the processing. The AP is in charge of guaranteeing the device's integrity as part of the MISC protocol.

The several extra chips that may be found on a medical device, such as actuators that communicate with the patient and sensors that take measurements, are represented by the Components. The AP is what keeps the device's integrity intact for the other components.

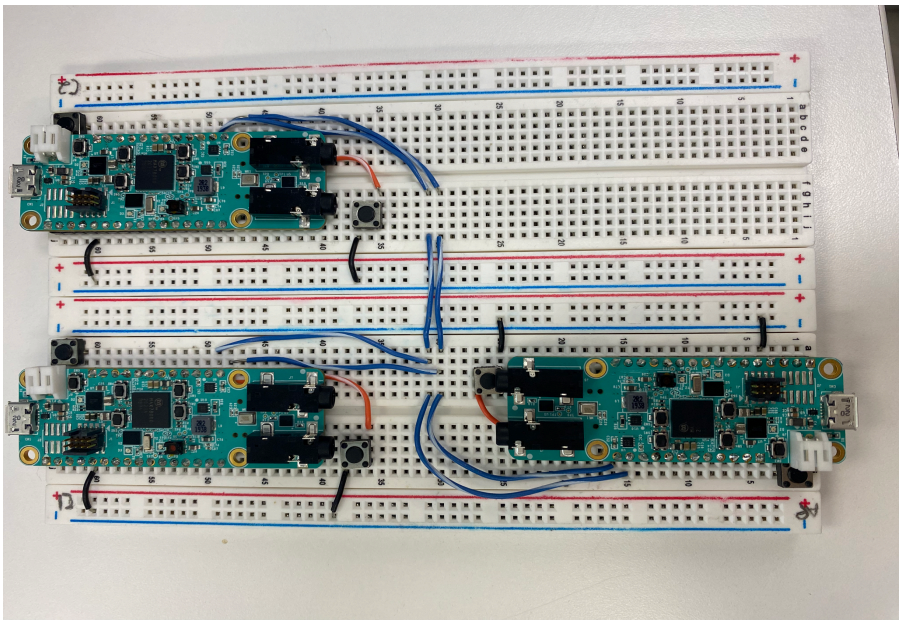

Figure 2 *eCTF Breakout Board*

Attack Phase Group Assignment

---

2. **Set-Up**

Initially, we installed Nix and Poetry Shell on our macOS computer. However, during the installation of Nix, an error occurred due to unrecognized path variables. To rectify this, we needed to edit our system's path variables manually. We proceeded to utilize Git for version control within our development workflow. Git enables efficient collaboration, version tracking, and code management across projects, ensuring seamless integration with our chosen development environment and tools.

For our development environment, we utilize Visual Studio Code (VSCode), incorporating essential extensions:

- C/C++
- Python
- Cortex-Debug

Additionally, for debugging purposes, we installed OpenOCD and arm-none-eabi. To ensure seamless functionality, it was imperative to add path variables specifically for arm-none-eabi.

Nix shell serves as an isolation environment, enabling the segregation of packages from the main system. This feature ensures a clean and independent space for software installation and execution. On the other hand, the Poetry shell functions as a dependency environment tailored for the Python programming language. It facilitates the management and execution of dependencies installed from Python packages, ensuring seamless integration and execution within projects.

During the attack phase, we employed the code provided by two schools: UTArlington and CA codes. After acquiring their code from the designated links, we proceeded to flash it onto our boards. This step enabled us to simulate and analyze the behavior of their implementations, facilitating a comprehensive assessment of potential vulnerabilities and security implications.

3. Attack Phase

Upon cloning the code from the provided link, our process entailed the following steps:

1. Building the deployment
   `ectf_build_depl -d ./`
2. Constructing the Application Processor
   `ectf_build_ap -d ./ -on ap --p 123456 -c 2 -ids "0x11111124, 0x11111125" -b "Test boot message" -t 0123456789abcdef -od build`
3. Developing the Component with our customized implementation
   `ectf_build_comp -d ./ -on comp1 -od build -id 0x11111125 -b "Component boot" -al "Poisened" -ad "04/04/24" -ac ""`
4. Flashing the boards
   `ectf_update --infile build/ap.img --port /dev/tty.usbmodem1404`
   `ectf_update --infile build/comp1.img --port /dev/tty.usbmodem1414`
5. Utilizing the Boot Tool to verify communication between the application processor and components.
   `ectf_boot -a /dev/tty.usbmodem1404`

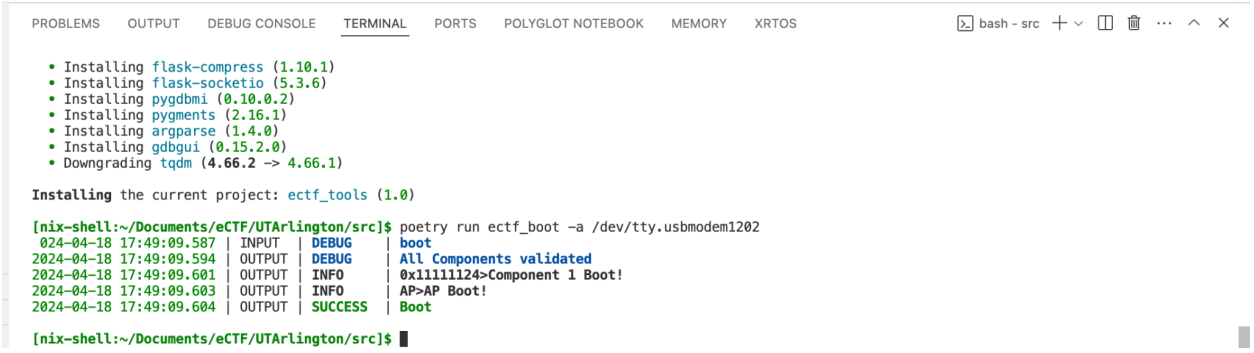

Figure 3 *Screenshot of the Terminal After Executing the 'boot' Command*

Figure 3 showcases a terminal screenshot post executing the 'boot' command. The terminal output highlights the successful booting of UTArlington components alongside the Application Processor (AP), affirming their seamless communication. Notably, the component's functionality has been compromised as a result of code manipulation in 'component.c'. This alteration

intentionally disables the LED flashing mechanism upon successful boot, signifying deliberate tampering with the component's behavior.

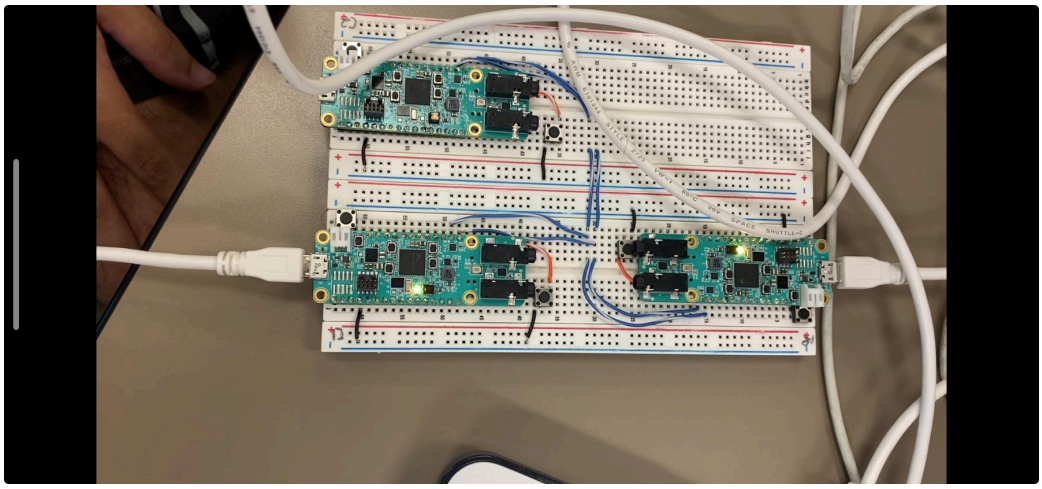

Figure 4 *AP communicates with the poisoned component, demonstrating system resilience*

Figure 4 illustrates the Application Processor (AP) successfully establishing communication with the poisoned component. Despite deliberately tampering the component's functionality, the AP effectively communicates with it, as evidenced by the terminal output. This interaction validates the system's resilience despite the component's compromised behavior, highlighting the robustness of the communication protocol and the AP's ability to adapt to altered component states.

We replicated the same steps for the code of CA, but unlike the previous case, it didn't validate the poisoned component.

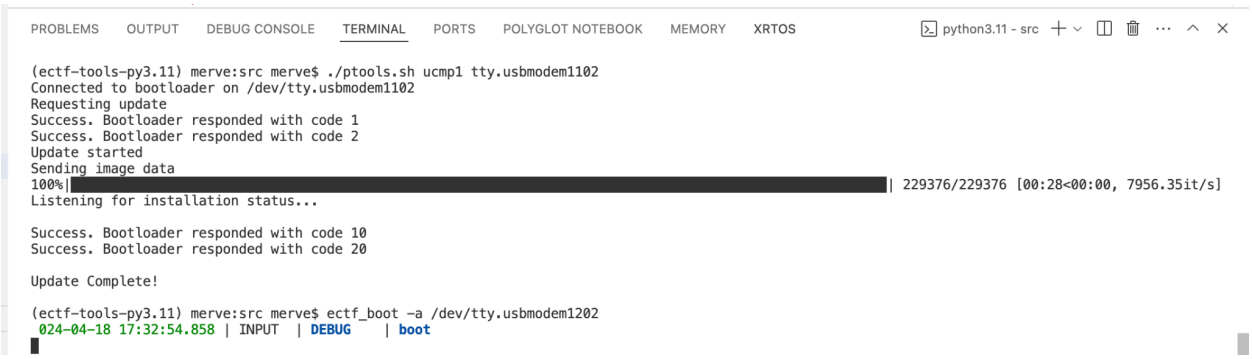

Figure 5 *Screenshot of the Terminal After Executing the 'boot' Command*

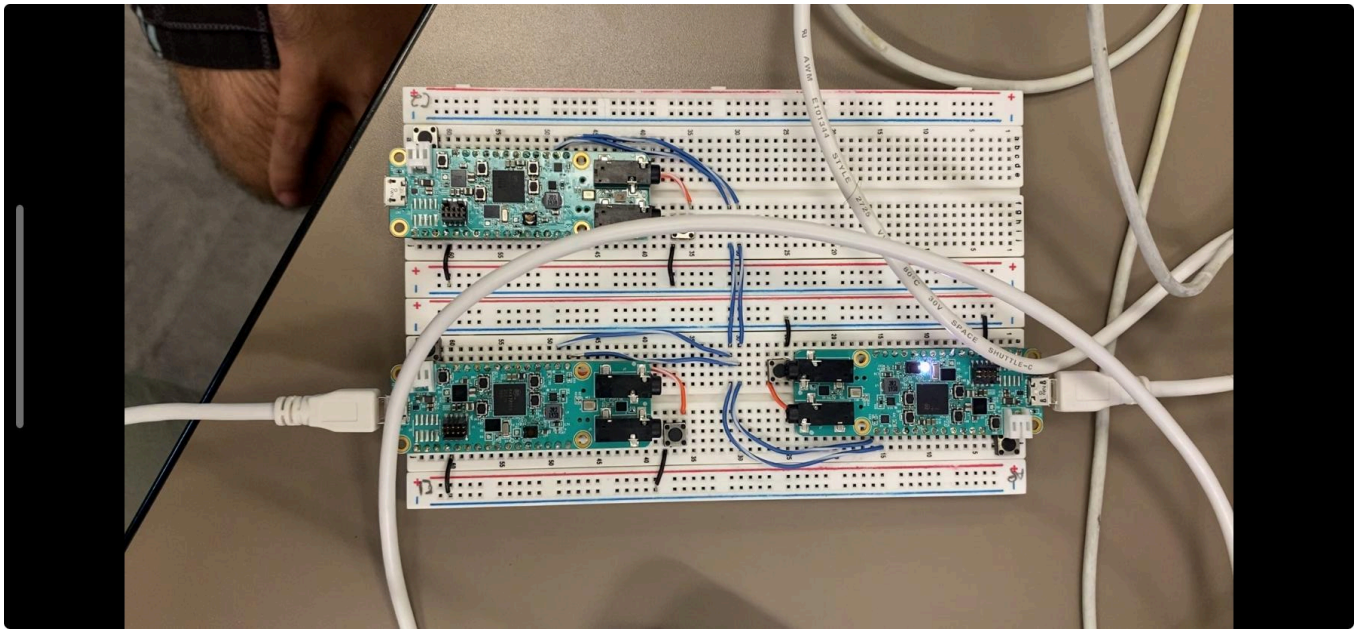

Figure 5 *AP doesn't communicate with the poisoned component, demonstrating system resilience.*

In conclusion, UTArlington's code has been proven it is not secure because it gets validated after the component has been modified, but CA does not validate the component after it has been modified it stays at the 'boot' flag and does not return any new messages from the bootloader. Therefore, we found that the CA code is secure while the UTArlington code is not.

This scenario is a supply chain attack, where the security compromise originates from within the development or distribution process of the software components. In this case, the compromise occurs during the development or deployment stages of the UTArlington code. By intentionally tampering with the component's functionality, an attacker could inject malicious behavior into the system, potentially compromising its security. This highlights the importance of robust security measures throughout the entire software supply chain to mitigate the risk of such attacks.

Attack Phase Group Assignment

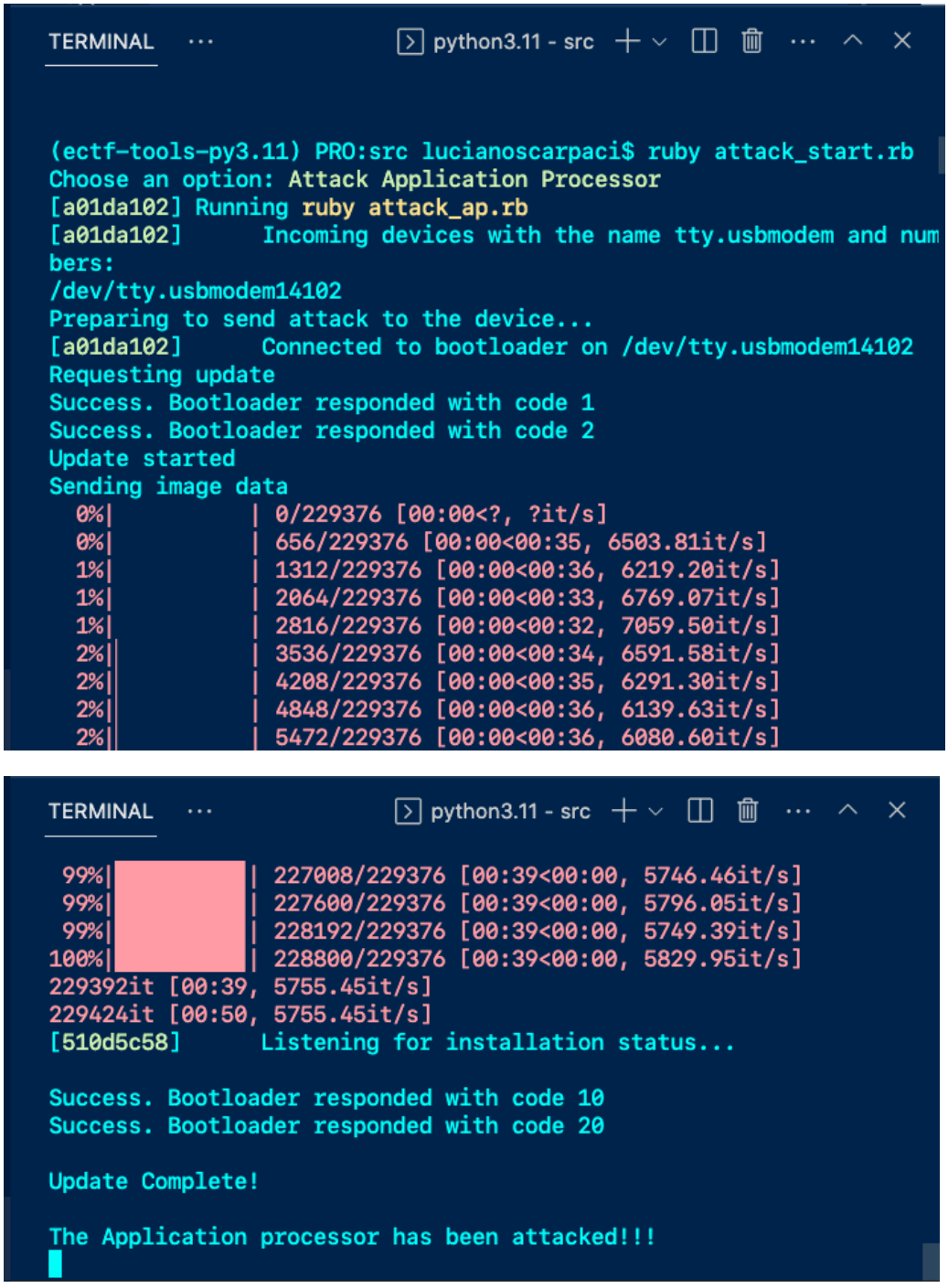

Figure . Attack UTArlington's Application Processor with Ruby

Attack Phase Group Assignment

This screenshot shows the Application Processor has been flashed with a ap.prot from the firmware/operational/ directory using a customized Ruby script that has a menu that allows the attacker to select if they want to attack the Application Processor.

Attack Phase 3: Supply Chain Poisoning

The components communicate to the application processor. The components have a secret and public information. A technician has two components A and B. Component B went bad. The attacker replaces components with their own components. The system will be booted with the poisoned component. We will use a modified version of their component. We used UTArlington code to build an Application processor. The code was successfully flashed. Now it is time to create a poisoned component. To create a poisoned component we need to find where in the component the ID is exposed using the debugger.

Figure . build the component packaging tool.

Our component has been built on UTArlington's deployment. Now it's time to debug the component to see any information that we are not supposed to see. We forgot to flash the board before debugging it. So the component was flashed with the c11111124.img.

Attack Phase Group Assignment

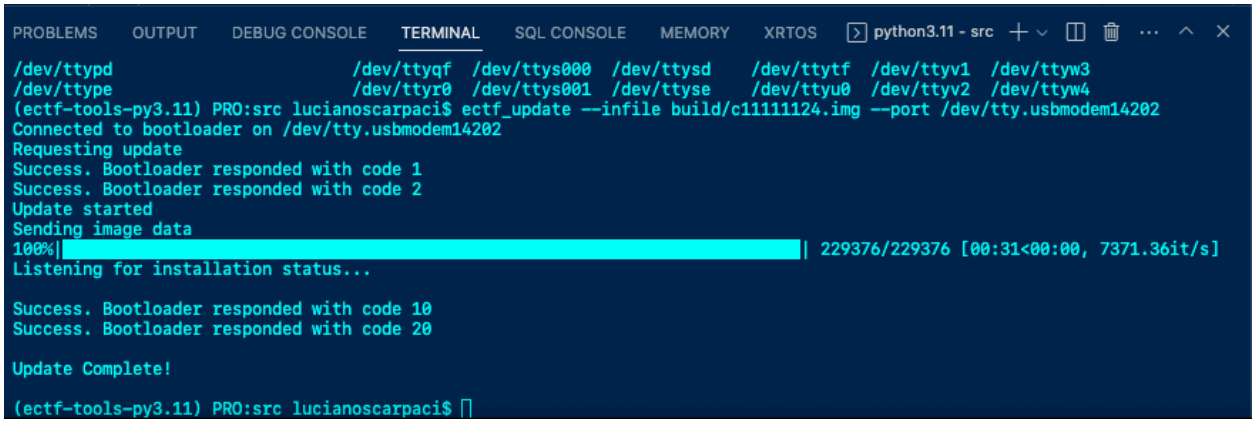

Figure. Flashed the component tool

Here is a screenshot of the component board with a green light showing a successful flash.

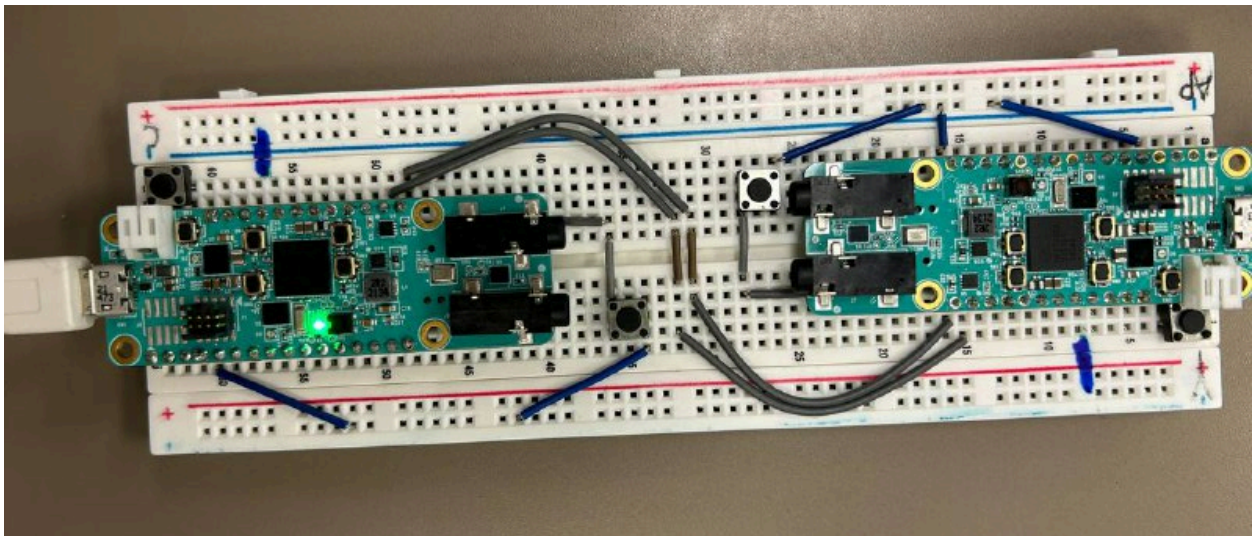

Figure. The component after being flashed with UTArlington's code

The solid green light indicates the component is working and has been flashed properly.

Attack Phase 4: Black Box

First I started with a fresh AP with insecure.bin. I went to the src/ directory of TTU and tried building the AP. Second, I noticed that the compiler was giving an error that the MSDK was missing so I cloned it from https://github.com/analogdevicesinc/msdk and put it in the src/ directory. The second error is that I had to rename the root directory from "Attack Code" to

"AttackCode". Third, I went to rerun the build command and I got wolfSSL errors. I thought there had to be a way to debug where the errors from wolfSSL might be coming from so I ran the build command with pdb and used the s and n commands to go to the next lines in the build code. This was working only while it is in Python. Once it leaves python then there is no debugger anymore so i cannot fix the error. Next I had an idea to install WolfSSL, so I went to the src/application_processor/wolfSSL directory and followed the instructions on INSTALL.
I ran the ./autogen.sh which checks the system to get ready to build wolfSSL. next i run ./configure and the make command which checks if i have g++ compiler on my system.

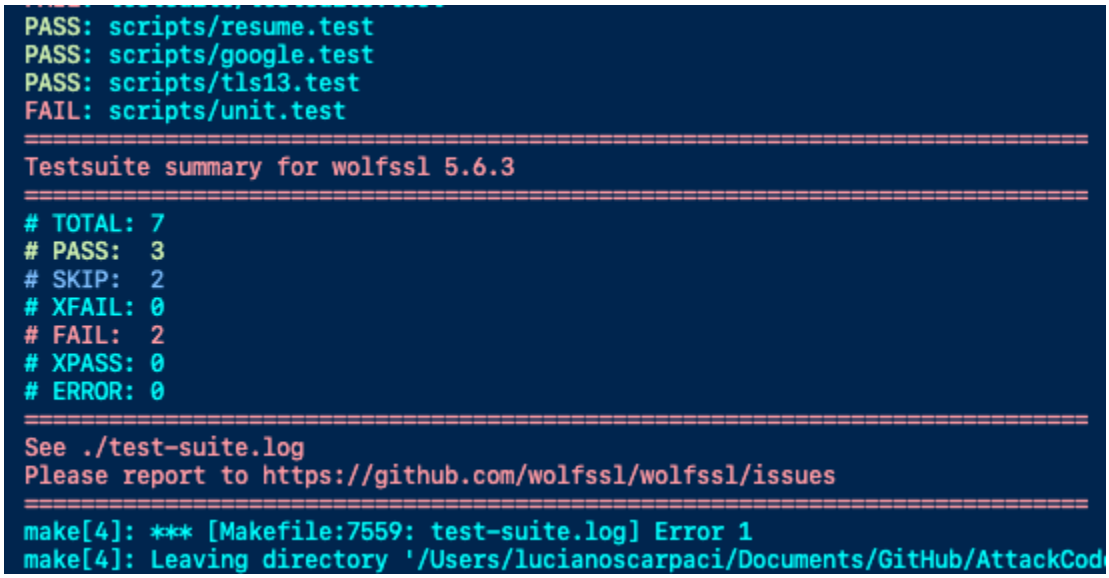

Figure. wolfSSL software testing tool

In total, the wolfSSL passed 3 tests and failed on 2 tests. After installing wolfSSL, I went back to the src/ directory to build the ap.bin. If the team built the AP correctly it will show a build file. In the directory firmware/blackbox/c0.prot is a binary file that needs to be attacked. To do analysis on this file, I used Hopper disassembler.

Attack Phase Group Assignment

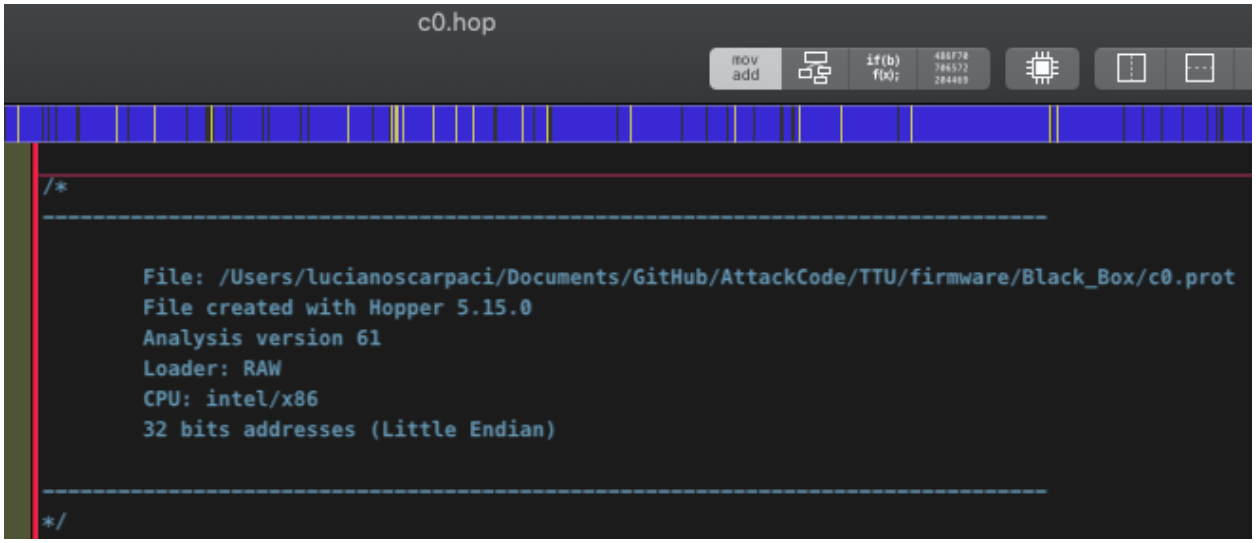

Figure . Hopper disassembly processing tool

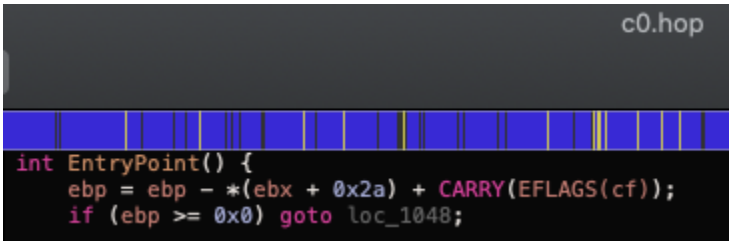

Figure . Hopper pseudocode analysis tool

In this code the EFLAGS is the hashed component ID 0xca9d9064 which is passed to the function loc_1048() because it is greater than 0x0. At function loc_1048() it performs many operations on the eax register.

Attack Phase Group Assignment

---

4. **Group Members**

- Luciano Scarpaci
- Merve Karabulut
- Hadise Pishdast
- Maryam Taghi Zadeh

5. Conclusion

The Embedded Capture the Flag (eCTF) competition, facilitated by MITRE Engenuity, served as a challenging yet instructive platform for honing our skills in creating secure embedded systems. Our team's central objective revolved around programming firmware for the MAX78000FTHR board, a task that demanded meticulous attention to security protocols and robust coding practices.

Throughout the competition, we leveraged various tools and methodologies to streamline our development processes and enhance system security. Adopting Nix for consistent build environments provided a standardized foundation, while Git facilitated efficient version control and collaborative coding efforts. Visual Studio Code (VSCode) emerged as our primary development environment, augmented with essential extensions like C/C++, Python, and Cortex-Debug for comprehensive debugging capabilities.

In the attack phase, we meticulously analyzed and scrutinized code provided by participating schools, namely UTArlington and CA. By simulating their implementations and conducting thorough assessments, we uncovered significant disparities in security vulnerabilities between the two codesets.

Upon manipulating the component's code to introduce intentional vulnerabilities, we observed contrasting outcomes in the validation process. While the UTArlington code validated the modified component despite its compromised state, the CA code exhibited resilience by abstaining from validating the poisoned component.

Ultimately, our findings underscored the critical importance of robust security measures in embedded systems development. The secure design and implementation of firmware are paramount in safeguarding against potential exploits and ensuring the integrity of critical systems. By identifying and addressing vulnerabilities through rigorous testing and evaluation, we contribute to the ongoing advancement of embedded security practices, ultimately fostering a safer and more resilient technological landscape.