CSAW 2023 Final Phase Report

Konstantinos S. Mokos
Department of Informatics
University of Piraeus
Piraeus, Greece
kostasmkuni@gmail.com

Ilias Fiotakis
Department of
Electrical Engineering
Democritus University
of Thrace
iliasfio@gmail.com

George Mamidakis
Department of
Computer Science
University of Makedonia
mamidakisgeorge@gmail.com

Meletios Michail
Department of
Electrical Engineering
National and Technical
University of Athens
melmichail@gmail.com

***Abstract*—This document outlines the response to the challenges published for the final phase of CSAW 2023 Embedded Security Challenge. In particular this report consists of a presentation of general conclusions drawn from the analysis of all the challenge sets, the methodology approach followed by our team and research performed for each task. The report is structured in 3 main sections, introduction, methodology and finally challenges where we provide the necessary evidence along with the steps followed towards the solution for each challenge with a brief description of them.**

***Index Terms*—Side Channel Attack, Timing Attacks, Reverse Engineering**

I. Introduction

Following the published instructions, our team utilized external hardware, including a Saleae logic analyzer and additional Arduino boards, to analyze and comprehend the serial communication protocols employed for each challenge and execute the attacks. We also utilized open-source reverse engineering tools for static and dynamic analysis, to better understand the attack vectors whenever the provided side-channel leakages did not provide a clear path on how the system operated. Reverse engineering of firmware implementation is a common attack surface that can be utilized to gain additional knowledge about the system, particularly when the hardware—such as the target board in question—hasn't been secured robustly, allowing for the firmware on the microcontroller's flash memory to be read easily.

Since all the members belonged to different universities, we were not able to meet in person. As a result, we explored efficient methods to perform virtual meetings throughout the entire duration of the competition. Our remote meeting setup included a Discord Server where important files such as code snippets and instructions on building copies of the target board were shared, as we will see in Section, II

II. Methodology

In order to maximize the efficiency of our team we crafted a methodology to approach each challenge. With this methodology we were able to perform a thorough examination of each problem at hand. This ensured that we had identified all the possible exploitation ways and selected the most optimal in each case. This process can be seen in Figure 1.

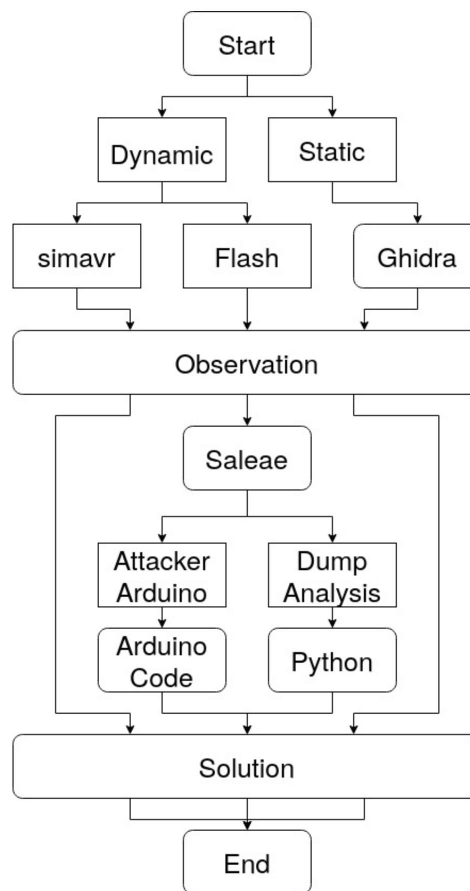

Fig. 1: Methodology Flow

According to the attack flow, presented in Figure 1, the first step of every challenge was to load the `.hex` file into our Arduino target boards and just interact with the binary and observe how the hardware responds to user-provided input. After we have concluded our initial assessment, we converted the IntelHex files to binaries and loaded them to Ghidra, an open-source reverse engineer tool, that also supports the AVR architecture, and its ability to produce a decompiled version of the binary. Simultaneously, we created several logic dumps using an analyzer of all signals present on the board's peripherals to be able and analyze them with better precision. Manually and visually inspecting the signals was also feasible in most cases.

A. Dynamic Analysis Environment

The primary dynamic analysis tool we used for different challenges was a logic analyzer, with which we captured the logic data sent from the Arduino to the peripherals. Another tool we used to gain a better understanding of the binaries runtimes, without using it to solve challenges, was simavr. With it, we could simulate the binaries and start a gdb server. After connecting to the simulated device, we would open avr-gdb, and connect to the exposed port on localhost, and begin debugging. Some of the challenges we tackled with debugging were checked for the peripherals, specifically the keypad. We would set a breakpoint to the relevant address, and manually change the return value to bypass the check. Though we spent some time trying to find a solution to somehow emulate the keypad, we didn't find anything before the end of the competition. Another tool

B. Replicating The Target Board

As previously mentioned, since the team members couldn't meet physically, and we were provided with only one board, we decided for the rest of the members to purchase the necessary sensors and components of the target board and replicate it ourselves. To identify the components, we looked at the supplied video to try and gain an early advantage and purchase them early. After searching for each sensor we waited for the board to arrive and one team member mapped all the connections to each peripheral using a multimeter. After a bit of manual testing and wire planning, we constructed 2 similar boards so we could all load the firmware and understand what was going on.

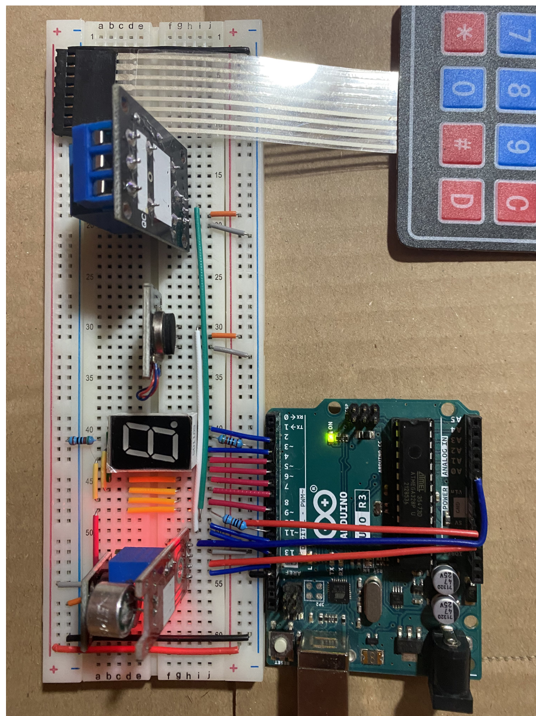

Fig. 2: Target Board Replica

C. Coding Framework

The coding solutions we chose were a mixture of Python and Arduino code. Python was chosen for a variety of reasons. It allowed us to quickly and easily analyze and process information from dumps and other files we generated, as well as to interact with the device and perform the attacks required due to the seamless communication it offers to devices through the serial library. Arduino code was used in attacks and scenarios where we used another Arduino to perform various attacks like brute-forcing and man-in-the-middle that required physical triggers. While moving through the challenges of the final phase we realized that we tended to reuse some specific code snippets for performing specific attacks, interacting with devices over serial, and many more. This led to the creation of a set of wrappers (callable functions) to automate specific processes. Investing time into this particular task proved beneficial since it increases the productivity and work flow during the development of our solutions.

III. Challenges

The section is comprised of three subsection, one per challenge set provided during the competition. Each of the subsections contains a summary box for each challenge accompanied by a challenge description consisting of a brief explanation of the code, the vulnerability in it and the exploitation method followed to achieve each challenges goal.

A. Week 1

All White Party

```
Binary Logic
Attack type: Timing Analysis
Challenge solution:
Barry - ?
```

`Challenge Description:` You and your friends have just arrived at the exclusive Hollywood "All-White Party", but you're missing invitations. You found a missing badge outside the entrance gate, but after scanning at the gate, the security system is asking you for a username and 10-digit password PIN credentials. Can you uncover the secret passcode, blend in with the glamorous crowd, and find a way inside the party to experience the glitz and glamour in a reasonable amount of *time*.

`Problem Statement:` Perform Timing Analysis to recover name, and conduct static and dynamic analysis to obtain information concerning the hash.

`Analysis:` This challenge had two parts. The first part asked for a username over serial, and the second one for a 10-digit code from the keypad. To recover the username, we performed a timing analysis on the response time of different input characters. The longer it took to print the 'Invalid username' message, the more characters we successfully identified. This vulnerability is due to the way the character check is implemented in the binary, checking one character at a time.

After we scripted and ran our exploit, we found the username. Then we were asked for a 10-digit code through the keypad. Trying random values, we got back that the SHA hashes do not match, and then some values are printed. After performing some attempts, we saw that the values printed occasionally changed, though there were only a few of them.

```
import serial
import time
import string

printable_characters =string.printable
response_time_avg =2.20

io =serial.Serial('/dev/ttyACM0', 115200)

def find_username():
   user_input =""
   char_count =36
   for _ in range(len(printable_characters)):
      user_input +=printable_characters[char_count]
      io.write(user_input.encode())
      start_time =time.time()
      response =io.readline().decode()
      print(response)
      end_time =time.time()
      response_time =end_time -start_time
      print(response_time, user_input)
      if response_time >1.05 *response_time_avg:
         response_time_avg =response_time
         char_count =0
      else:
         user_input =user_input[:-1]
      time.sleep(0.01)
      char_count +=1

def welcome():
   while True:
      response =io.readline().decode()
      if 'serial):' in response:
         break

try:
   welcome()
   find_username()
except KeyboardInterrupt:
   io.close()
```

As for the hash, we attempted multiple approaches. Performing static analysis on the binary, we found through yara rules that the SHA hashing algorithm used is SHA1, and identified some parts in the binary which used them, like FUN_code_000afc. We also found them in offset code:000067. Going through the binary, we identified multiple parts of the binary's logic, like the string pointers for the printed messages, the check of the keypad, the decoding of the keypad i2c input, and the comparison of the hash bytes. But due to the complexity of the code and hard readability of the decompilation, what we could achieve was restricted.

One of the problems we faced while performing dynamic analysis was we couldn't simulate the keypad. One of the approaches we tried was to write an i2c slave to emulate the keypad. This was implemented by using an extra arduino board. Though it was infeasible for our setup, we could theoretically bruteforce all the possible 10-digit numbers with this method and we understood how the i2c protocol works completely.

```
#include <Wire.h>

int button_press_send =0;
bool first_send =false;
byte byte_to_send =0xE0;

void setup() {
   Wire.begin(0x20);
   Wire.onReceive(receiveEvent);
   Wire.onRequest(requestEvent);

   Serial.begin(9600);
}

void loop() {
   delay(10);
}


void receiveEvent(int howMany) {
   while (1 <Wire.available()) {
      char c =Wire.read();
      Serial.print("c: ");
      Serial.println(c, HEX);
   }

   int x =Wire.read();
   Serial.print("x: ");
   Serial.println(x);

   if(x ==0x0F) {
      byte_to_send =0x0E;
      Serial.println("first");
   }

   if(x ==0xF0) {
      byte_to_send =0xE0;
      Serial.println("second");
   }

   if(button_press_send) {
      byte_to_send =0xF0;
      button_press_send =0;
   }
}

void requestEvent() {
   Wire.write(byte_to_send);
   Serial.print("sending: ");
   Serial.print(byte_to_send, HEX);

   if(byte_to_send ==0xE0 and !first_send) {
      button_press_send++;
      first_send =true;
   }

   if(byte_to_send ==0x0E) {
      button_press_send++;
      first_send =false;
   }
}
```

Another approach was to load the binary in a debugger and set breakpoints in different addresses to check the internal state of the memory, and to bypass different check. Through this we found out that one of the set of numbers printed on a wrong input, 167 98 212 144 128, are the first five bytes of the SHA1 hash of the username, Barry. Since it checks only the first five bytes, we performed an exhaustive search of all SHA1 hashes and their first five bytes to see if we could find any matches

with the known bytes that are returned. Unfortunately, this didn't return any promising results.

```
Bluebox

Binary logic: Phone Simulator
Attack type: Frequency Analysis
Challenge solution:
B339B009
```

Challenge Description: In a secret underground lair, you and your team uncover the hardware for legendary blue box hack. To unlock the secrets of the lair, you must decode telephone frequencies and recreate the iconic audio tones in order to reveal the flag. Its time to unearth the blue box technological history before the authorities arrive to shutdown your operation.

Problem Statement: Analyze the DTMF-like keypad frequencies and map the transmitted tones into the alphanumeric values.

Analysis: Our initial task, like the rest of the challenges, was to load the firmware onto the board. Flashing it on the board triggered the buzzer to play four tones. Upon connecting to the serial interface with screen, we discovered that we need to replicate these tones one by one.

We began conducting research based on the challenge's title, which led us to explore phone phreaking and related attacks. Our initial assumption was that we might need to perform a blue box attack, but that path turned out to be a rabbit hole. After experimenting with the keypad, we made educated guesses about how this challenge operated, and we realized that each key on the keypad was associated with a specific frequency.

To address the first part, our approach involved capturing the frequency of each key using a tool called 'Sonic Visualizer' while pressing each key individually. We started with '1', then '2', and continued in sequence. It's worth noting that every four key presses generated a new four-tone sequence as part of the challenge. However, we chose to ignore it until we had mapped all the keys on the keypad. After adjusting the settings in 'Sonic Visualizer', we ultimately obtained the following results:

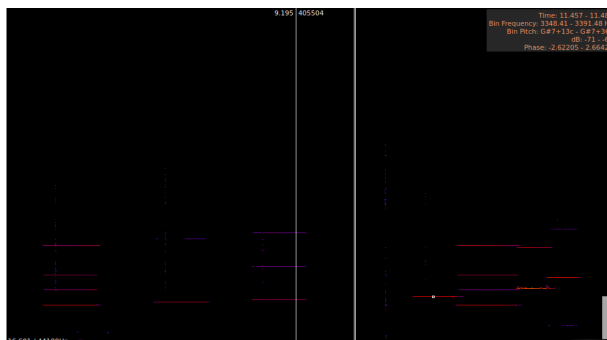

Fig. 3: Frequencies loaded to Sonic Visualizer

```
1: 2800 - 2900 Hz
2: 3000 - 3100 Hz
3: 3100 - 3300 Hz
A: 3300 - 3500 Hz
4: 3500 - 3600 Hz
5: 3700 - 3800 Hz
6: 3900 - 4000 Hz
B: 4100 - 4200 Hz
7: 4200 - 4300 Hz
8: 4400 - 4500 Hz
9: 4600 - 4700 Hz
C: 4800 - 4900 Hz
*: 5000 - 5050 Hz
0: 5100 - 5150 Hz
D: 5500 - 5550 Hz
```

An interesting note to consider here is that the last frequencies were not precisely defined. It's possible that the laptop's microphone was not optimal, resulting in a recording with various frequencies that required analysis. Fortunately, we were able to distinguish them effectively.

After successfully mapping all the keys, we recorded the next four-tone sequence, and by examining the frequencies, we were able to recreate it. The challenge then played the flag, and we realized that to decode it, we needed to repeat the process of translating each tone into a letter. After experimenting for a while, we determined that the flag was "B339B009". We tested it with the keypad, and we were rewarded with the happy song.

We also attempted to automate this process by using the script we had created during the "allwhite" challenge, where we managed to simulate the keypad with an Arduino. However, this approach proved to be extremely time-consuming, so we decided to focus on finding a solution for the second part of the "allwhite" challenge, which unfortunately remained unresolved.

B. Week 2

```
Operation SPItFire

Binary logic: Custom Message
Transferring
Attack type: Dynamic Analysis
Challenge solution:
SPyBURNd
```

Challenge Description: Amid the digital battleground, you, an accomplished spy, are assigned the mission of deciphering an intricate maze of wire traffic acquired from the mysterious hacker collective, SPItFire. To assist your efforts, your remote team has gained access to a device linked to one of SPItFire's surveillance camera, allowing you to clandestinely exchange messages. Your objective: find out how to communicate with the security camera, and acquire to

uncover the coveted password flag that can be used to infiltrate their security footage.

Problem Statement: Analyze the transmitted "HELLO" message with a logic analyzer, and the errors received from the binary to understand how to send the FLAG message.

Analysis: When we started the challenge, we observed that as soon as the message 'Receiving message "HELLO"' appeared, the arduino started sending 5 Volts to the relay, indicating a message was being sent. To capture that, we connected a logic analyzer to it. When we opened the capture in Logic 2, we tried to apply analyzers like SPI, but nothing worked, so we tried retrieving the values by observing the data bus.

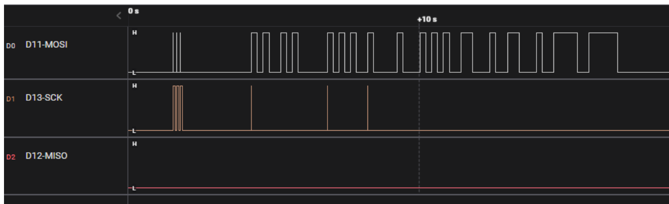

Fig. 4: HELLO logic capture

Through this process, we got the bits 10100101 00000101 01001000 01000101 01001100 01001100 01001111 00111110, which correspond to the hex value a50548454c4c4f3e. This decodes to HELLO, along with some other interesting values. To understand those values, we focused our attention to the challenge program.

Running the Linux command strings on the binary, we saw that 4 different errors existed inside the binary, 'Incorrect Header', 'Incorrect Length', 'Length must not exceed 10', and 'Bad CRC'. Running the binary and entering FLAG in hex, that is to say 464c4147, we got the incorrect header response. Since a5 was the first value in the hello message, we tried to prepend that to the flag message. Doing so returned the incorrect length error. With this we confirmed that a5 was the header byte. The next byte we thought would be the length byte. Again, looking at the HELLO message, 05 is before the HELLO message, and it is exactly the length of it. However sending a504464c4147 still produced incorrect length. Since the above message has one extra byte, we appended a random byte to our message, and got the bad CRC error. This indicated that the last byte was the CRC byte. Since we got only one byte, we assumed that the algorithm was CRC8. We found the FLAG CRC8 byte to be da, and after appending it to our message and sending it, we got the 'Recieving Flag... message'.

Since the flag was again transmitted through the relay, we connected our analyzer and captured the flag. One error we faced was the inconsistency of the CRC8 byte, which can also be seen at the HELLO message, since it is 0x35, not 0x3e. We attempted to read the data and bruteforce the various CRC8 byte, but we could not settle on a correct answer. To resolve

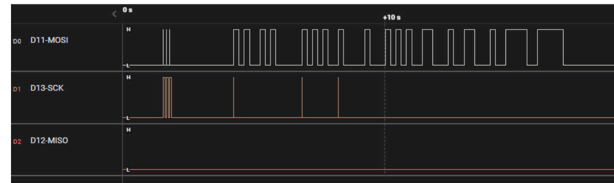

Fig. 5: Correct flag logic capture

our issues, we exported the flag captured data as CSV, and parsed them.

```
import csv

file_path ='digital.csv'
time_info =[]
time_diff =[]
temp_time =0
start_bit ="1"
with open(file_path, 'r') as file:
   csv_reader =csv.reader(file)
   next(csv_reader)
   for row in csv_reader:
      time =float(row[0])
      state =int(row[1])
      time_diff.append(abs(temp_time-time)*1000)
      temp_time =time
      time_info.append(time)
time_diff =time_diff[2:]

bits =[]
for time in time_diff:
   time =round(time)
   n_bits =round(time/200)
   bits.append(n_bits)

flag =""
for bit in bits:
   flag +=start_bit*bit
   if start_bit =="1":
      start_bit ="0"
   else:
      start_bit ="1"

def binary_to_ascii(binary_string):
   n =8
   chunks =[]
   for i in range(0, len(binary_string), n)
      chunks.append(binary_string[i:i+n])
   ascii_string =''
   for chunk in chunks:
      ascii_strings +=chr(int(chunk, 2))
   return ascii_string

ascii_output =binary_to_ascii(flag)
print(ascii_output)
```

czNxdTNuYzM (s3qu3nc3)

```
Binary logic: Math and Frequencies
Attack type: 7-Segment Display
Decode
Challenge solution:
97349616
```

Challenge Description: A cryptic dance of numbers unfolds before the eyes. Can you harmonize with the rapid rhythms of this challenge? In this realm of numbers, a symphony can conquer even the swiftest of mysteries.

Problem Statement: Decode the 7-segment display Saleae dump.

Analysis: Upon flashing the firmware into the board, the 7-segment display began to exhibit various characters. We promptly connected a logic analyzer to capture all the transmitted data. Additionally, upon inspecting the serial output, we discovered a hint: QSBzb3ByYW5vIG9mIHNvdW5kLCByZWFjaGluZyBmb3Igd GhlIGhlYXZlbnMu, which decoded into: 'A soprano of sound, reaching for the heavens.'.

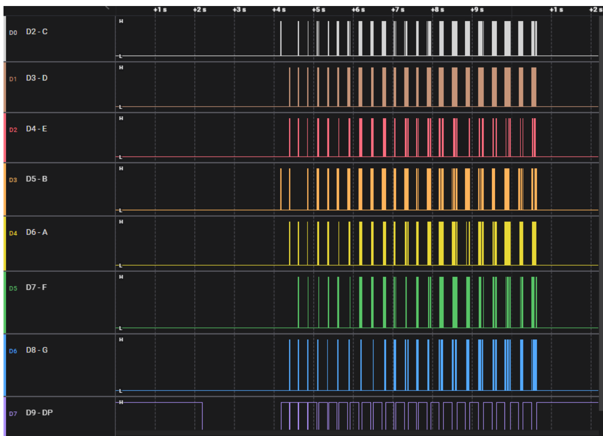

Fig. 6: 7 segment display logic capture

To analyze the logic dump, we used Python. Initially, we extracted the data into an svg file, and subsequently, we utilized Python to parse it. The 7-segment display essentially consists of 8 inputs, which are correspondingly mapped to the letters a, b, c, d, e, f, g, and h.

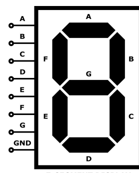

Fig. 7: 7-segment display channels

Our script collected all the active letters at a specific point in time and converted them into ASCII text, which was then displayed on the 7-segment display.

```
f =open('digital.csv', 'r').readlines()

flag =""
link ={
      0 :'c',
      1 :'d',
      2 :'e',
      3 :'b',
      4 :'a',
      5 :'f',
      6 :'g',
      7 :'h',
}

codes =[]
last =''
last_batch =[]
for line in f[1:]:
   signals =line.split(',')[1:]
   signals[-1] =signals[-1].strip()
   data.append(signals[0])
   data.append(signals[1])
   data.append(signals[2])
   data.append(signals[3])
   data.append(signals[4])
   data.append(signals[5])
   data.append(signals[6])
   data.append(signals[7])
   code =""
   cc =0
   for char in data:
      if char =='1':
         code +=link[cc]
      cc +=1
   codes.append(code)
   last_batch.append(code)
   if data =="0"*8:
      print(last_batch)
      max_length =max(last_batch, key=len)
      flag +=max_length #.replace('h','')
      flag +=" "
      last_batch =[]
   last =code
print(flag.replace('h ', ''))
```

The result appeared as follows:

cb debag cdeafg debag d cbfg cb d
debag d cdebaf debag d cdebaf cb d
cbfg d cdebaf cb d cb d debag d
cdebaf cb d cdebaf d cdebaf d
cdebafg d cdebaf cb d cdebaf d
cdebaf d cdebafg cb d cb d cdebaf d
cdebafg d cdebafg cdbafg d debag d
cbfg cb d debag d cdebaf d cb d
debag cdebafg d cdafg d cdebafg cb d
debag d cdebafg d cba d cdebaf debag
d cdebaf d cdafg d cdbafg d debag d
cdebaf cdbag d cdafg d cdebaf d
cdebaf d cdeafg d cbfg d cdebaf cb d
cdbafg d cbfg d cbfg d cdebafg d
cdebaf cdbag d cdeafg d cdbafg d cdafg
d cb d debag d cdebaf cb d cdebafg d
cbfg d cba d cdafg d cdeafg cdbag d
cdebafg d cba d cdbafg d cdebafg d cba
d cdeafg cb d cba d cdeafg d cdbag d
cdafg d cdebafg cbfg d cdebaf d cdafg
d cdeafg d debag d cdbag d cbfg

Which was decoded using dcode.fr into:

```
126_41_2_02_01_4_01_1_2_01_0_0_8_01_0_0_81_1_0
_8_89_2_41_2_0_1_28_5_81_2_8_7_02_0_5_9_2_03_5
_0_0_6_4_01_9_4_4_8_03_6_9_5_1_2_01_8_4_7_5_63
_8_7_9_8_7_61_7_6_3_5_84_0_5_6_2_3_4
```

We also categorized the data based on the pauses we observed in the dump so the final version was something like:

```
1
2
6
2_4
1_2_0
2_0
1_4_0
1_1_2_0
1_0_0_8_0
1_0_0_8
1_1_0_8_8
9_2_4
1_2_0_1_2
8_5_8
1_2_8_7_0
2_0_5_9_2_0
3_5_0_0_6_4_0
1_9_4_4_8_0
3_6_9_5_1_2_0
1_8_4_7_5_6
3_8_7_9_8_7_6
1_7_6_3_5_8
4_0_5_6_2_3_4
```

After analyzing the resulting numbers, it became evident that they formed a Fibonacci-like sequence. Our attempts to analyze them through scripting proved futile.

```
output =[1, 2, 6, 24, 120, 20...]
```

Following some extensive debugging, we resorted to online research. Searching both the name of the challenge and the numbers led us to this website. Remarkably, the sequence on this website perfectly matched the one from the challenge. When we tried the next number in the sequence, we successfully obtained the flag: 97349616.

For a more in-depth examination of the binary, we imported it into Ghidra and attempted to locate any relevant information. While we identified numerous intriguing logics and frequently used memory addresses, such as DAT_mem_024c, the decompilation proved to be quite intricate. Nonetheless, we did manage to identify the code responsible for conducting the check.

C. Week 3

The following subsection outlines a general analysis of the challenges contained in set 3. Due to limited amount of time our team did not have the time to implement all of the attacks and exploitation methods that will be described for each challenge. However, we still achieved to perform important analysis and note down ideas of how we would continue.

Sock and Roll

```
Binary logic: Locked Factory Door
Attack type: Brute force frequencies
Challenge solution: -
```

`Challenge Description:` In the heart of the whimsical and colorful Happy Socks factory, you find yourselves caught in a captivating and slightly bizarre world. This factory is a maze of vibrant rooms filled with the magic of sock-making machines, piles of socks adorned in all shades and patterns, and, of course, the joyful, dancing sock mascots that are the emblem of this fantastical place. However, what was initially a delightful visit has turned into an unexpected challenge. You have been mysteriously locked inside one of the factory's rooms, with no apparent way out. The laughter and cheerfulness of the factory have given way to a sense of urgency as you realize they need to escape. Luckily, you find the Happy Tap Dancing Socks Message Machine 2000, which seems to be transmitting some strange message. The only way to exit this colorful world and return to reality is to send a distress signal using this cutting-edge messaging technology.

`Problem Statement:` Create a frequency generator to brute force different frequencies in order to find out what the distress signal is.

`Analysis:` The challenge seemed to revolve around a messaging system that operated over the air and utilized two of the target board's peripherals: the buzzer and the microphone. The buzzer generated two alternating tones with frequencies around 32 kHz and 1 kHz, representing an "all-good" message. Our objective was to transmit a similar message in the same format but as a distress signal.

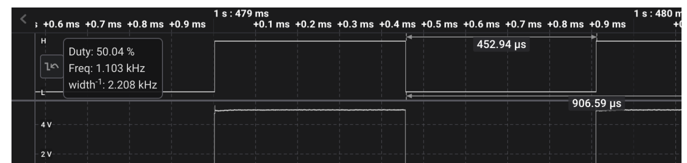

Fig. 8: The 1kHz signals

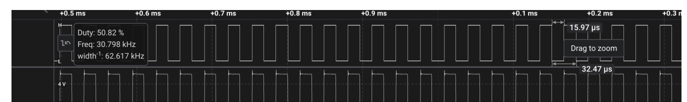

Fig. 9: The 31 kHz signals

Our analysis primarily focused on capturing a digital signal from the buzzer using a logic analyzer. We used the timing

information to extract details about the transmitted signal, including its duration, frequencies, and the number of alternations per message transmission. Subsequently, we attempted to generate other signals following the same format using Python and the sounddevice library. Unfortunately, we were unable to identify any valid patterns or obtain additional information about the messaging format.

```
import numpy as np
import sounddevice as sd

# Parameters for the square wave
frequency =1100 # Frequency in Hertz (A4 note)
duration =2.0 # Duration in seconds
sample_rate =44100 # Sample rate in Hertz

# Generate the time values
t =np.linspace(0, duration,
int(sample_rate *duration), endpoint=False)

# Generate the square wave
square_wave =np.sign(np.sin(2 *
         np.pi *frequency *t))

# Play the square wave
sd.play(square_wave, sample_rate)

# Wait for the tone to finish playing
sd.wait()
```

Furthermore, when analyzing the analog input signal from the microphone, it appeared that no data was being transmitted while the buzzer was in operation. However, removing either the buzzer or the microphone resulted in a message transmission failure, leading us to believe that these two components were operating as an emitter-receiver duo.

In our last attempt, we conducted an exhaustive search in terms of frequencies, akin to a brute force attack, to identify the appropriate message. Due to the overwhelming sound generated by the buzzer and the sound produced by the Python code, we attempted to directly feed the signal into the microphone. Unfortunately, our efforts were in vain, as we could not obtain any clear logic captures while the target board was operating normally.

Vender Bender

```
Binary logic: Motor Movement
Attack type: Fault Injection
Challenge solution:
mMmCaNdY
```

Challenge Description: You roll up to that there vending machine, and it's making a soft hum, like a well-tuned engine, promising you a sweet snack for your taste buds. You eyeball the colorful snacks, deciding if you want them chips for a salty crunch or that chocolate bar for a sweet fix. You've got some coins clinking in your hand, ready to drop 'em in and get that engine running. You pause, and think if there is a better way... a free way. Maybe if you trigger an error before the snack is dispensed, you can get your money back? Can you pump the brakes when your hear the gears whirring, and make off with your snack like it's a freshly greased wrench? Then lets take those taste buds on a delicious test drive.

Problem Statement: Perform a simulated fault injection attack to the vending machine's motor.

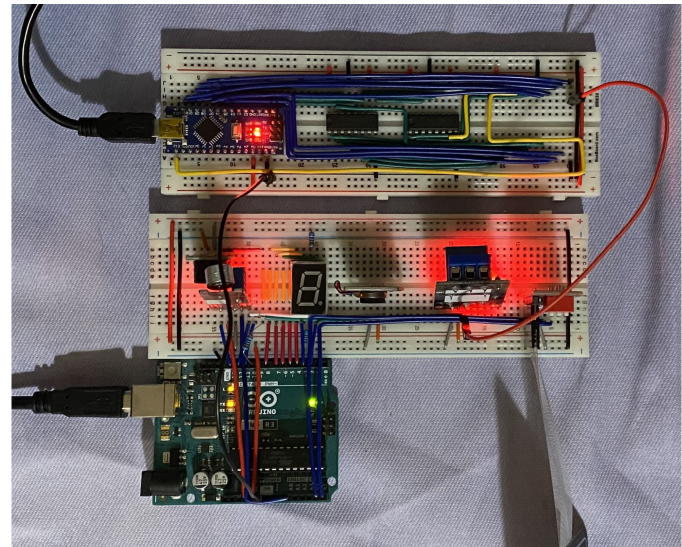

Fig. 10: Exploitation setup

Analysis: Upon loading the firmware onto the board, the relay began toggling. Connecting to the serial interface revealed two recurring messages. The first message read: 'After receiving credit, send "ERR" to jam the motors' while the second message stated: 'Motor movement SUCCESS. A snack was dispensed for $2. Insert another credit for a new snack.'

We recognized that we need to send the "ERR" message at a specific time and began doing it manually. Unfortunately, manual attempts yielded no success.

To automate the process, we wrote a script to send the "ERR" message and induce an error. After numerous attempts, we generated a few motor 5902 errors, but determining the correct timings remained elusive.

```
import serial
import time
import string

printable_characters =string.printable
response_time_avg =10

offset =0
FAULT_KEY ="ERR".encode()

# Define the port and baud rate
ser =serial.Serial('/dev/ttyACM0', 115200)

try:
   ser.write(FAULT_KEY)
   response =ser.readline().decode().strip()
   print(response)
except KeyboardInterrupt:
   ser.close()
```

We then documented each instance of relay toggling and the timing of messages in the serial output. Our observations revealed a crucial detail: there existed a narrow window of approximately 2 seconds when the relay toggled, and nothing was printed in the serial output. It became apparent that the ”ERR” message needed to be sent during this brief window.

To be precise, the challenge followed this sequence:

- The relay remained closed for 3 seconds, with no activity in the first 2 seconds.
- Following this, the first message was transmitted, followed by another second of inactivity.
- The subsequent state was the ON state, lasting 7 seconds, following the same pattern as the OFF state. However, the second message was printed during the ON state.

Our deep understanding of the challenge was instrumental in our progress.

Initially, we sent an ”ERR” message after the relay toggled from zero to one, introducing a 3-second delay in the Python script, and then sending another ”ERR” message. This approach aimed to generate the motor failure, but the relay state toggling timings were not consistent. Some timings were too rapid to manage manually.

```
try:
    ser.write(FAULT_KEY)
    time.sleep(3)
    ser.write(FAULT_KEY)
    response =ser.readline().decode().strip()
    print(response)
except KeyboardInterrupt:
    ser.close()
```

As a result, we opted to write Arduino code and introduce additional hardware to the challenge. The concept involved detecting when the relay toggled and sending the ”ERR” message directly to the main board's serial interface.

```
#pushbutton connected to digital pin 7
int inputPin =8;
int previous_value =0;
# variable to store the read value
int value =0;

void setup() {
    Serial.begin(9600);
      # sets the digital pin 7 as input
      pinMode(inputPin, INPUT);
    # read the input pin
    previous_value =digitalRead(inputPin);
}

void loop() {
    # read the input pin
    value =digitalRead(inputPin);

    if(value !=previous_value) {
        previous_value =value;
        Serial.println("ERR");
    } else {
        Serial.println("HALT");
    }

    delay(100);
}
```

However, the serial communication between the boards disrupted the relay toggling, leading us to rely more on Python. We utilized an attacker Arduino to read the relay signal and print either ”ERR” or ”HALT” to the serial and the Python script was developed to monitor the serial data from the attacker board and send ”ERR” at the precise moment to the main board.

```
import serial
import time
import string

printable_characters =string.printable
response_time_avg =10

offset =0
FAULT_KEY ="ERR".encode()

# Define the port and baud rate
ser1 =serial.Serial('/dev/ttyACM0', 115200)
ser2 =serial.Serial('/dev/ttyUSB0', 9600)

try:
    while True:
        response =ser2.readline().decode().strip()
        if response =="ERR":
            print(response)
            ser1.write(FAULT_KEY)
            response =ser1.readline().decode()
            print(response)
            if "5/5" in response:
                for _ in range(60):
                    response =ser1.readline().decode()
                    print(response)

except KeyboardInterrupt:
    ser.close()
```

With this modified approach, we waited, and finally, the delightful tune that plays when obtaining the flag played.