**Abstract**

With greater strides of innovation in every sector of technology, we felt it was important to tap into the current situation of the products that a normal consumer buys. Since the subject in scope of our project includes security, the best consumer product we thought to look into was the IP camera. With an increasing demand for personal security, the IP camera was the perfect consumer product to dive into and see what is the security it offers. An Internet Protocol camera, also known as an IP camera or a WiFi camera, is an IoT Device that does not require a local recording device, just a local area network. With our project, we aim to look into the operating system of the camera we bought from an online marketplace, look into how easy is it to detect the communication protocol the camera follows, and how straightforward is it to gain a foothold in the operating system of the camera and access its files. Another aspect of checking how secure the software is of the IP camera is checking if it is possible to rever engineer the firmware the vendor provides. Reverse engineering is the process of breaking down the components of a previously made device, software, process or system to observe how the smallest of tasks are carried out. Reverse engineering a firmware entails that the attacker has complete control over the operations of the operating system the camera is working on.

Contributions from this project:

1. Getting a better understanding of how the different components work together on a PCB inside the camera
2. Mapping out the communication pathway the camera is on
3. Reverse engineering the IPC firmware that the vendor ships with the camera to gain a foothold on the operating system.

Detecting whom the camera is communicating to, and what files are accessible to an average consumer, plays a very important role in the advancement of the security manufacturers provide with everyday IoT devices.

# **Chapter 1**

**INTRODUCTION**

### **1.1 Chapter overview**

This chapter focusses on introducing the project by putting a consumer product into focus and testing it for the security measures that the hardware and the software provides.

### **1.2 The Problem**

In these times, security and privacy go hand in hand for everyday activities. People prefer having their eyes and ears on everything around them. And with increasing popularity of Internet of Things (IoT) devices, most the gadgets people buy nowadays are available for relatively cheap as well. This has led to a rise in the number of people buying surveillance cameras for their domestic purposes. These cameras, known as IP cameras (Internet Protocol camera, also known as a WIFI camera), have become a simple choice for anyone who would like to add an extra pair of eyes and ears in their surroundings.But when it comes to budget IoT devices, the only downside to these devices is that they don’t really get much firmware upgrades, which leaves a big hole in the security aspect in devices which have a primary role of storing data over a server.

### **1.3 Project Objective**

The goal of this project is to analyse the security measures taken by a budget IP camera, put into light what are the shortcomings to such devices when it comes to protecting the data a consumer is sharing with the device. All IP cameras are connected to the home network, the network which a consumer has setup at their places, but storing data in the cloud means that the camera is actively communicating to a server, and quite possibly sharing the data it stores in it as well, which, in this case, are the home network information, the audio and video footage. We are going to take a look into this problem and see what are the extents to which a person’s privacy is at large.

### **1.4 Project Proposal**

The goal of this project is to shed some light on how budget IoT devices handle the data they store. Hardware or software, where are the shortcomings to a budget IoT devices, because expensive IoT devices have something that the budget ones don’t have, which makes them very much more secure, and that is regular firmware upgrades. But because these easily available budget devices use outdated hardware, newer firmware upgrades are not relevant for them, leaving a big hole in the privacy these devices should be offering.

# 

# **Chapter 2**

**Problem Definition and Objectives**

### **2.1 Problem Definition**

To look into the security measures taken by a budget IP camera by looking into it’s hardware and software.

### **2.2 Scope and Objectives**

#### 2.2.1 Scope

Open up the camera to analyse the hardware

Connect the camera to a computer to analyse the firmware the vendor provides

#### 2.2.2 Objectives

Make a report on the security shortcomings a budget IP camera comes with, hardware and software

### **2.3 Hardware and Software Requirements**

#### **2.3.1 Software requirements**

1. Linux operating system

2. Reverse engineering software (GHIDRA and binwalk)

3. Wireshark

4. TeraTerm

#### **2.3.2 Hardware requirements**

1. IP camera

2. Computer

3. Soldering Kit

**Chapter 3**

**EXPERIMENTAL DESIGN**

### 

### **3.1 Chapter Overview**

This chapter aims to cover the design of the project. We will go over the structure of the tests we will be conducting in a systematic manner.

### **3.2 Introduction**

In this project, we are aiming to conduct a series of tests to try and expose the various security shortcomings in a budget IP camera in a methodical manner.

To explain some experiments, we will be using the help of simple diagrams for an easier visual understanding.

### **3.3 Plan of Action**

* The Teardown
* Setup and Network Analysis
* Gain access to the firmware and the IPC
* Reverse Engineering the IPC program
* Reverse Engineering the camera-server protocol

**Chapter 4**

**IMPLEMENTATION**

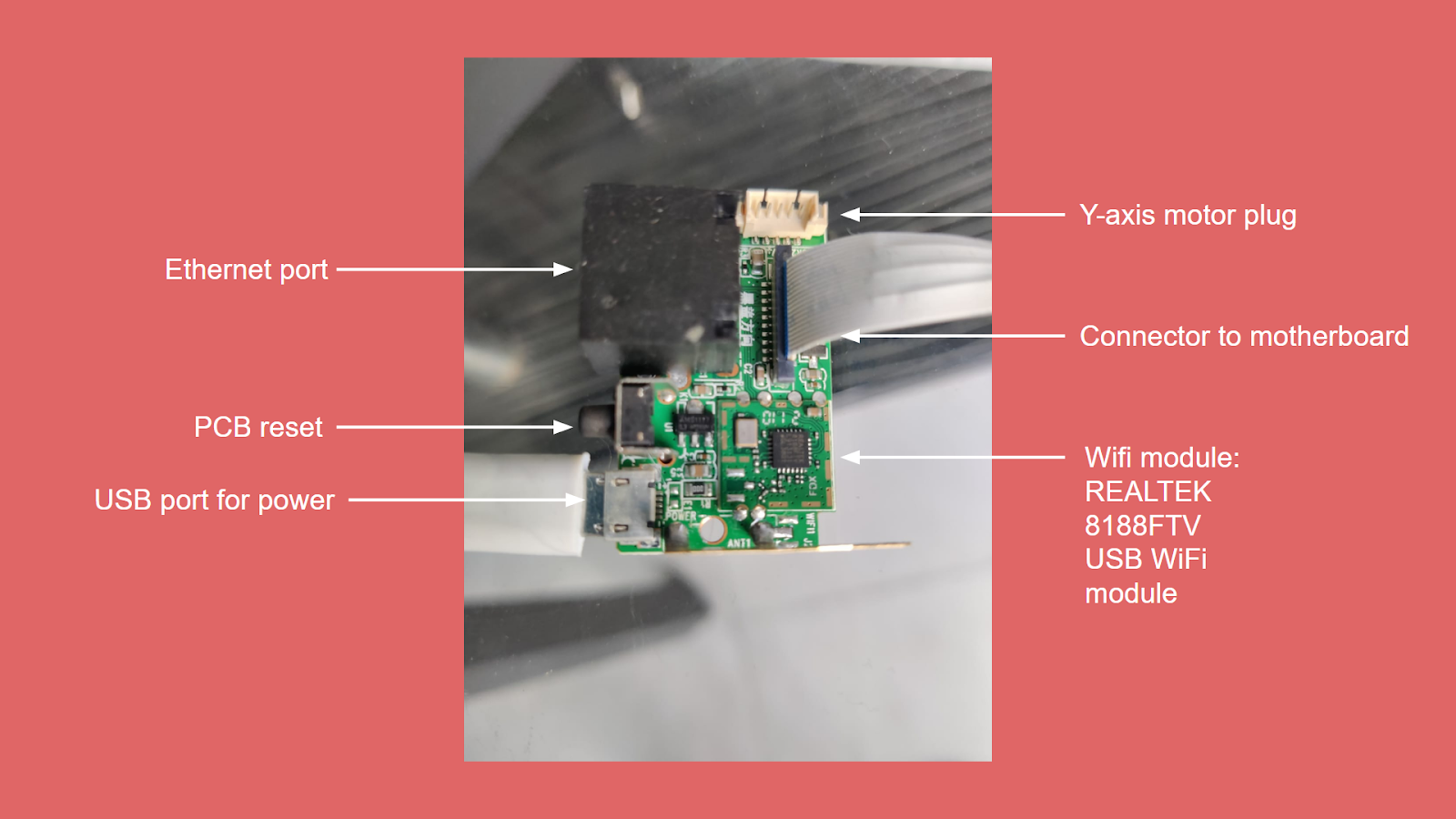
### **4.1 Chapter Overview**

In this chapter, we will be showcasing every single step we took to expose all the possible security shortcomings present in the camera we purchased. We will go about these steps one by one.

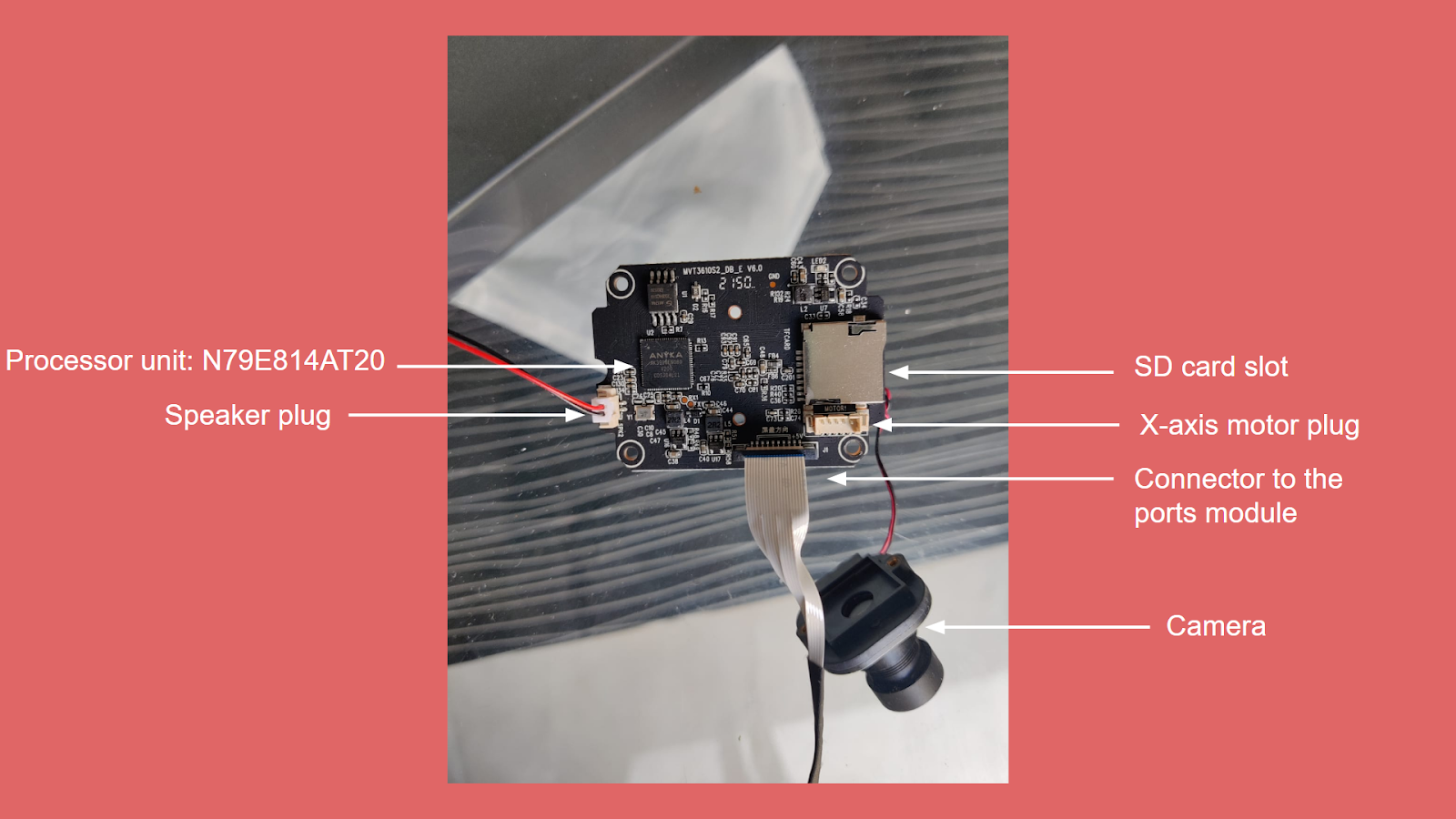
### **4.2 Discovering the Camera**

A “Nuvoton ARM9 SoC (N329x)” with an “N79E814AT20” CPU was discovered during the device's breakdown. A noticeable (and highly significant) micro switch beneath the glass dome allows the app to detect the camera during setup, and GPIO terminals are offered near to the CPU (Pin mask is 15).

The PCB connects to the motorised gimbal, which can rotate left and right but stops at 90 degrees on each side. This results in a total rotation of 180 degrees, which can be adjusted using API instructions.



802.11 or wired Ethernet are used for network connectivity, the Ethernet taking precedence.



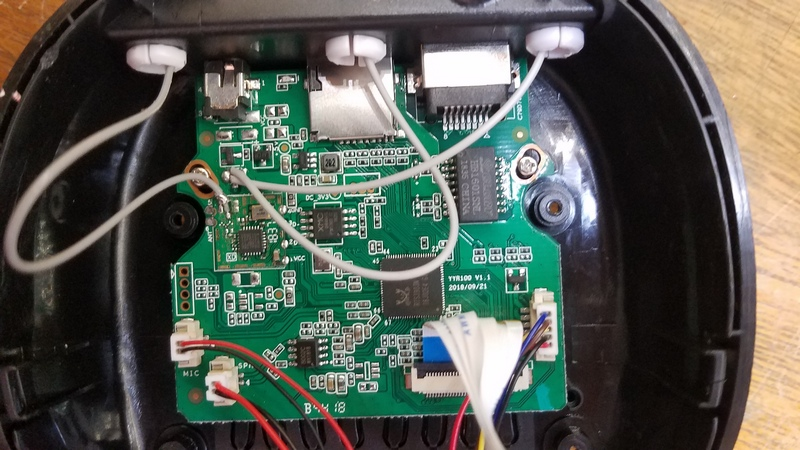
An LED on the camera indicates the state of the system and may be controlled with a custom command. When the camera is working/observing, the LED is constantly on, but it flickers when sending footage to a server or notifying any movement.

The “V380pro mobile app” is free and can be downloaded from the normal app stores; for this study, we downloaded the app from Google Play Store. The app serves as a hub for all of your V380pro cameras. Through the usage of a common API, the app is documented as the only proper way of communicating with the camera, and it supports many other IP camera models.

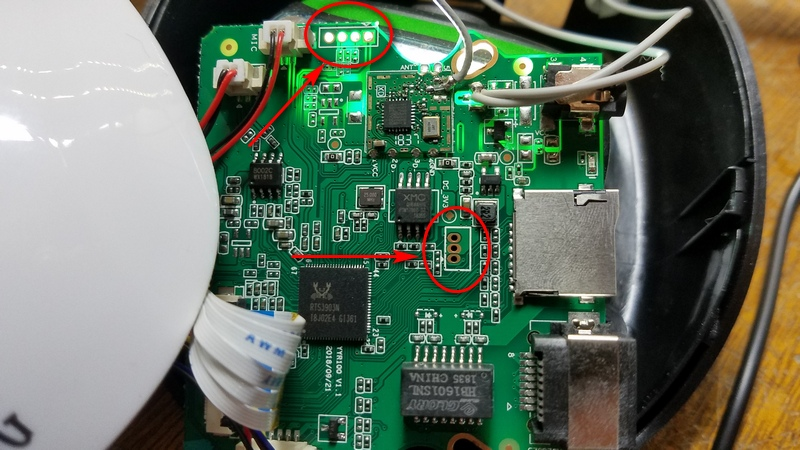
The only way to configure and operate the camera, according to the manual, is by installing an app in your phone. The configuration sequence is:

* Power up the camera. Since it is not yet configured, it will create an internet WiFi network, used only for configuration
* Launch the app on your phone. It will find the WiFi network created by the camera and connect to it
* The app will ask for your WiFi (home) network name and password. It will send this data to the camera
* The camera connects to the Internet using your WiFi and notifies the manufacturer server that it is alive
* The app connects also to the server and now you can control your camera

Apart from this, we also noticed there are 3 antennas, but two of them are connected to the same point on the PCB, as seen in the image below.

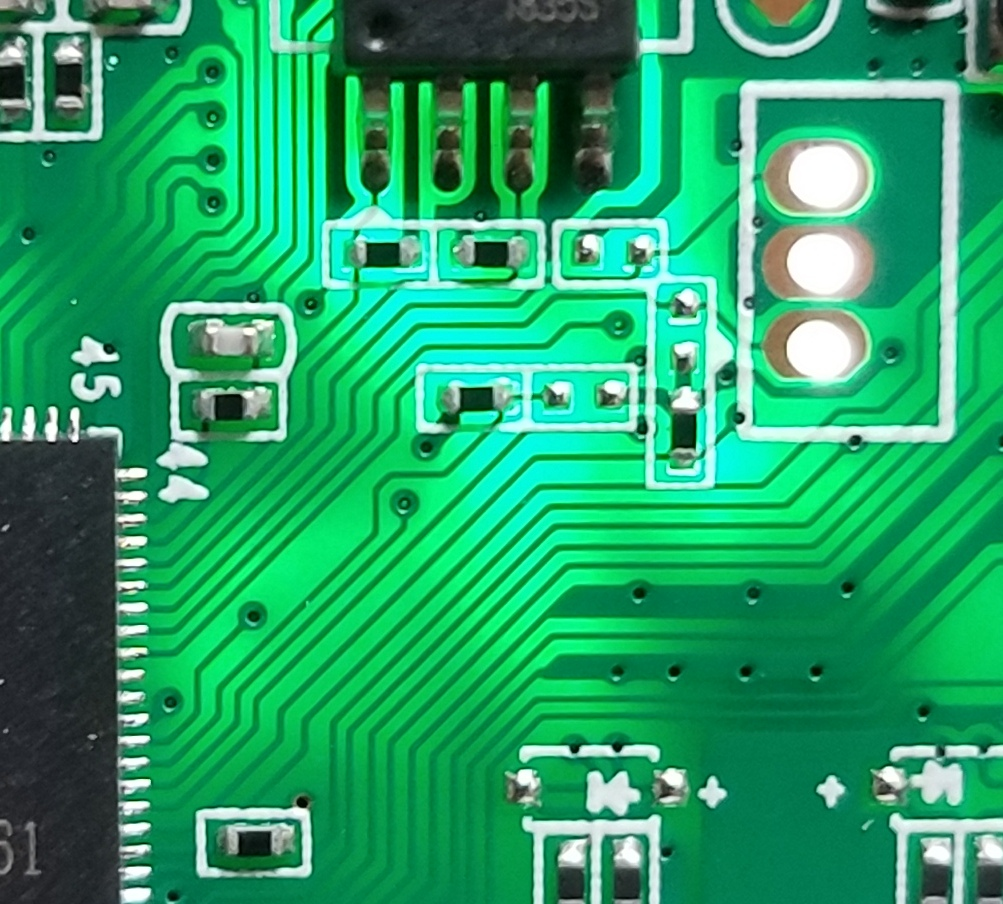


Here you can also see, there are two groups of unused pins that could be, maybe, debugports left by the developers. I’ve marked them in the next picture.



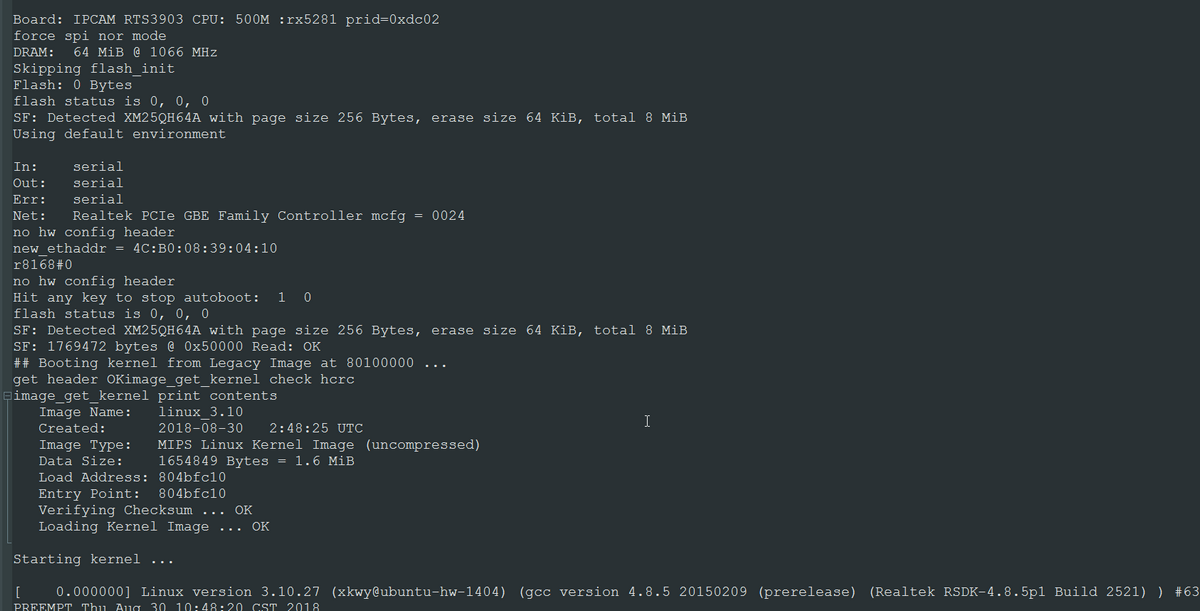
The first one, seen at the top of the image, appears to be connected to the antenna-attached second PCB. That's most likely a dedicated WiFi configuration processor. The second set of pins, on the other hand, appears to be connected to a huge CPU, which is most likely the PCB's main processor. Let me begin with this one.

It has three pins, thus for a serial port, we'd need at least RX, TX, and some power, most likely GND. I can see how they are related by shining a strong light under the board:

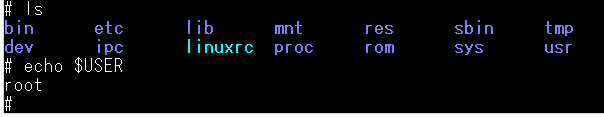


Now we can see clearly: The center connector is connected to the PCB ground. The other two are connected to the main processor.

Now, we could try to find out the data rate used by this port using an oscilloscope, but it is way easier to simply guess that on a terminal software. So after soldering these 3 pins to a cable, and connecting the newly attached cable to a computer running on linux, we launched a software called TeraTerm, a lot of things start appearing on the screen. It seemed that the camera was communicating with the computer.



I could type on my terminal connection when the camera finished its initialization phase. So maybe I could use the terminal connections to send commands to the camera. Let's try this:

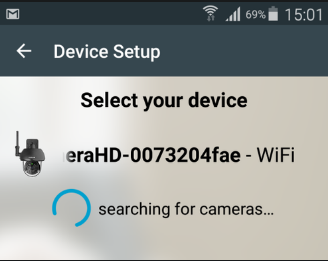


So yes, as it turns out, the operating system mounted on this camera just has one user, and that’s root, so we basically just got complete access to the file system, just like that.

### 

### **4.3 The Network Setup**

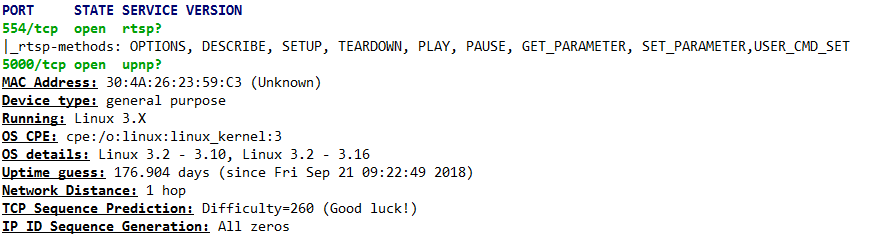
While setting up, we can either connect the Ethernet cable or go about pressing the camera’s pairing button, which switches the camera to host mode and enables a wireless network which is open. An open wireless network is just a plain way of indicating an insecure network. The software then searches for this network and asks the user to join it. This is a concerning feature for a camera designed for external use, especially since the camera also offers a number of unfiltered network services, such as a network video feed (RTSP), a customised internal messaging service for activating alerts, and two separate web servers (nuvoton and busybox), one of which has an unregistered firmware upgrade page.



When the app connects to this open access point, it sends queries to the nuvoton web server to run a wireless scan of nearby networks using the Linux iwlist command, with the results returned to the app as XML so you may choose your network from a list.

After entering the Wifi security key, it is then broadcast unencrypted over the open network, along with some rudimentary HTTP Authentication, like username ‘user’ and password '1234.' The query string is a strange amalgamation of the “SSID”, “PSK”, “username”, and “password lengths”, followed by the fields themselves.

This HTTP Authentication looks to be obsolete and is not in use; this is a condition we found to be very prevalent on this device; for example, there are numerous legacy webpages on the camera, such as /routersetup.html. A simple study of these reveals that this device was formerly used as a baby monitor. The fist thing is to find if the camera operating system provides a terminal connection, like SSH, so we could control the camera. Using nmap to see if there's any port open:



“nmap” founds 2 open TCP ports (554 and 5000), neither related to SSH. 554, however, is related to RTSP, a streaming protocol. That could be an option for receiving the camera images.

“nmap” also helps by identifying the operating system (Linux 3.X). That's also a good information, because now we know it uses an opensource OS, not any closed proprietary software that would make things way more complicated.

### **4.4 Analysing the Network Traffic**

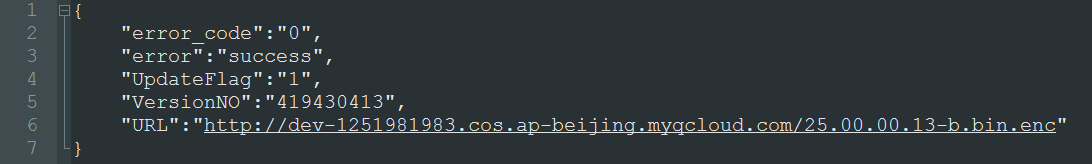
Because the camera uses our WiFi password to relay video and audio from our home to an external server, we believe we have a right to know what happens to this information. So we started Wireshark to examine the data sent and received during the camera setup process.

There's a lot to detest about this. I don't think the camera should be connecting to so many servers. I'm not sure why my camera needs to connect to Google, Alibaba, or Facebook, but it's at least attempting to resolve those addresses over DNS.

One interesting thing: The app informs that there's an update available for the camera firmware. So, probably, the camera figured out this by connecting to its manufacturer server, and the app is asking me if I'll allow the installation. By monitoring the traffic, I see the camera is checking this address:

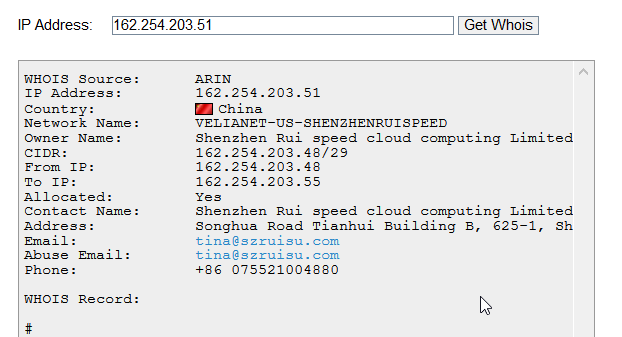
http://upg.cloudlinks.cn/api/device/checkupgrade.ashx?ApiVersion=1&DeviceVersion=419430406&KeyID=73dd8ec6718551f970cdcefd8ec77826&CustomerNO=0

Which returns the following json data:



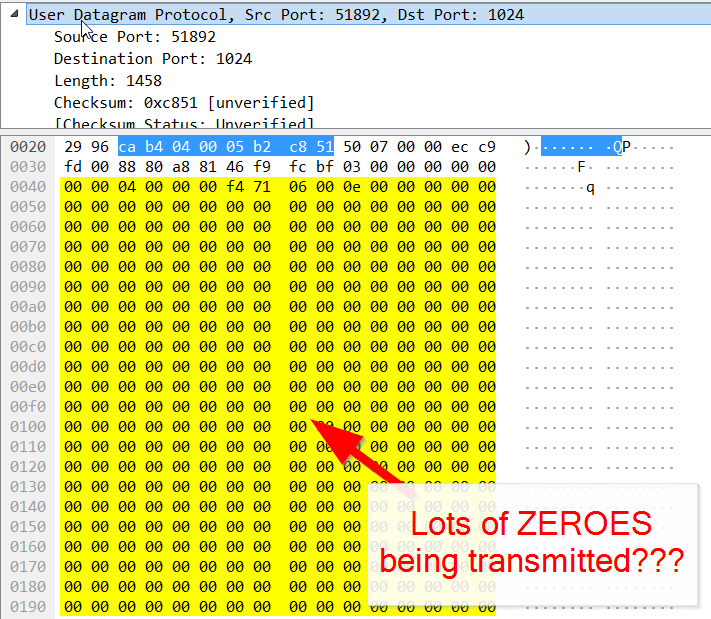
So that's the new firmware version's URL. It's simple to download for me. The firmware file is encrypted because the manufacturer does not want anyone spying on its software. Later, I'll try to figure out how to decode this. Let's return to the video feed.

On Wireshark, the server receiving my video is easily identifiable. A lot of data is delivered to this IP once the camera joins. When the camera is restarted, however, a different server is used. The camera most likely has a list of servers it can use and switches between them from time to time. This is one of them:



I'm not sure if I should be concerned about my video and audio being forwarded to China. I'm not sure, but I'd rather it wasn't transferred to any other country because I'm right next to the camera and it could send it to me immediately. So, let's keep going.

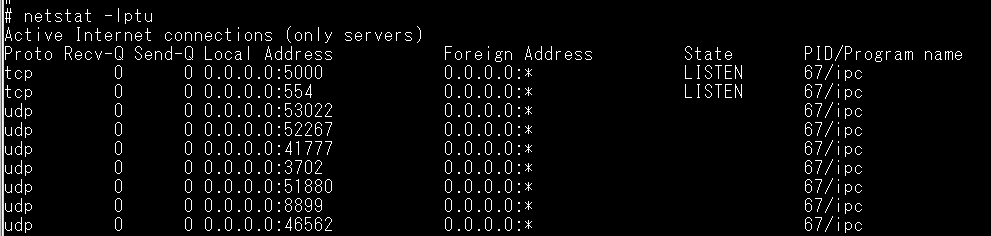
Data is originally delivered to a server listening on UDP port 8000 after configuration. This appears to be an initial connection; perhaps the camera and server are using it for a handshake. Since the largest packages go to this port, it appears the actual video is transferred to UDP port 51892. It appears to convey a lot of empty data, which is intriguing. Since we're sending video across the globe, I'd anticipate good data compression, however this protocol doesn't appear to be very efficient in terms of reducing network traffic:



### 

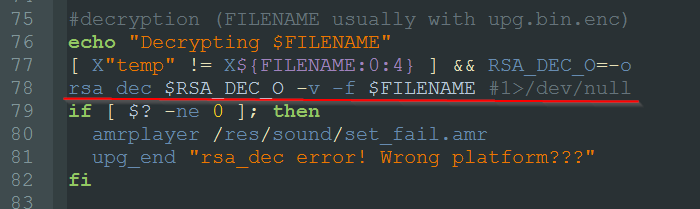
### **4.5 Extracting the Camera Softwares**

So, let's see which program is running the main camera tasks (connection to server, sending video stream, etc). I'll do that by checking which processes controls each internet sockets:



So, it seems to be a single process called ipc that listens on all these ports. I found the process file and, unfortunately, it's a binary file. I was hoping for a python script or something that I could check the code.

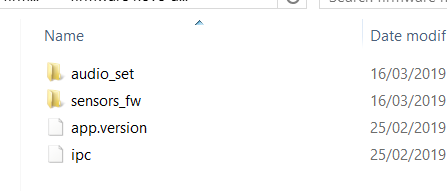
However, there's something I can still try. On Part 1 I found that the camera could download a firmware update via internet, but it was encoded. Now, with full access to the camera operating system, I could find the updater script. Here a piece of it



The marked line seems to be the code that decrypt the downloaded firmware. So I put the downloaded file on a SD card, inserted it on the camera, and run the command:



This extracts a binary which is probably a compressed file, so we copied it from the linux environment to our windows environment and tried extracting it with 7zip, and these were the contents:

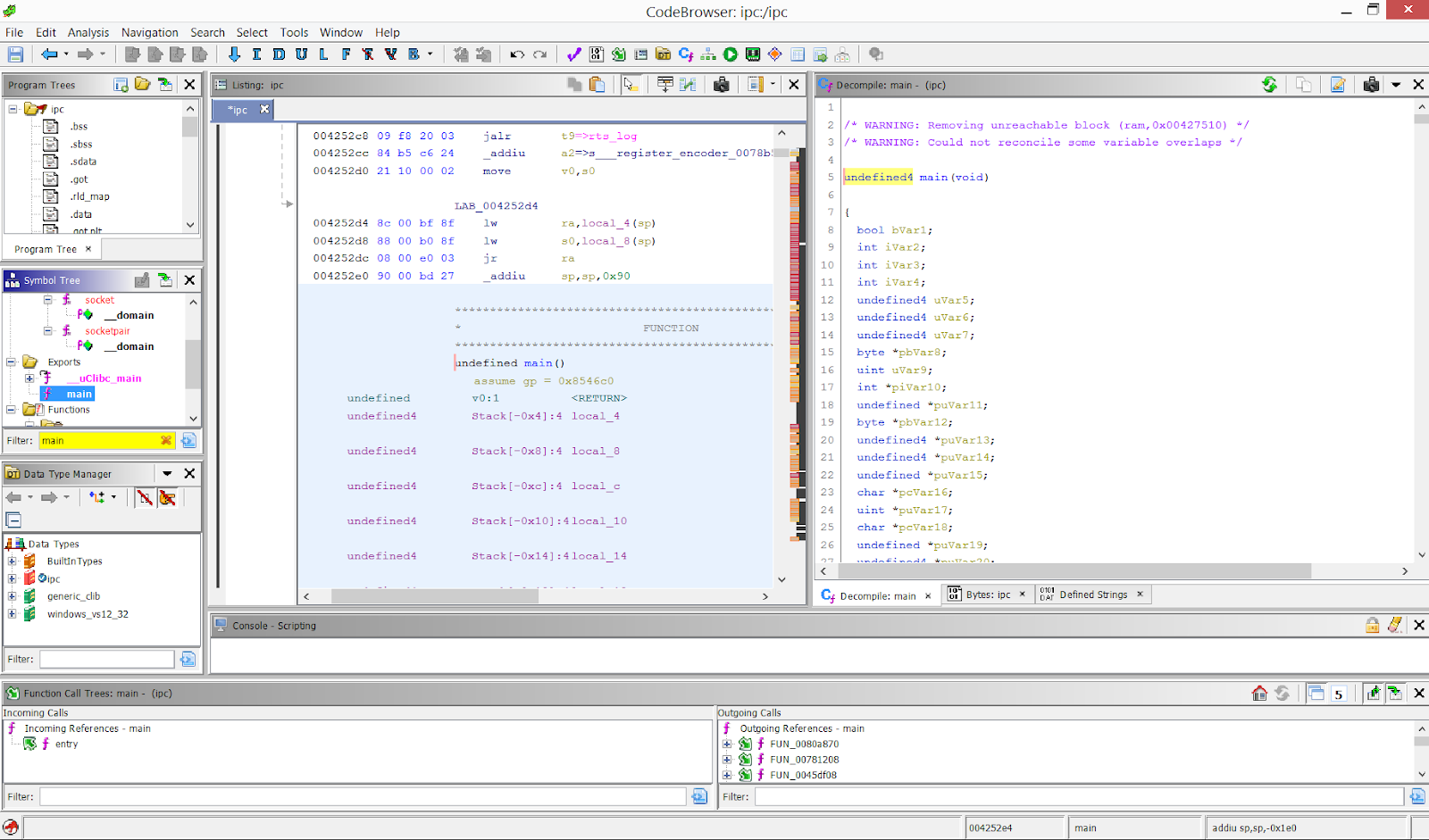


### **4.6 Reverse engineering the Camera firmware**

#### **4.6.1 The IPC-Program**

In order to analyze IPC I’m using Ghidra, a great reverse-engineering tool released by NSA. So, after loading and evaluating the IPC programme, this is the Ghidra CodeBrowser window. The main window in Ghidra is CodeBrowser, where we spend the most of our time looking at the disassembled code. Ghidra also has a fantastic decompiler that converts disassembled code to C code, which is really useful. I'll bear in mind that because I didn't know the specific processor to configure Ghidra, I couldn't put my faith in it 100 percent. But, for the time being, everything appears to be in order.

On the right side of the window we can see the IPC main() function:

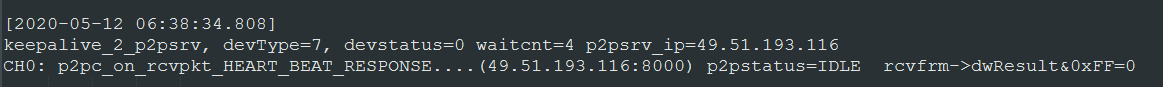


#### **4.6.2 Camera-server protocol**

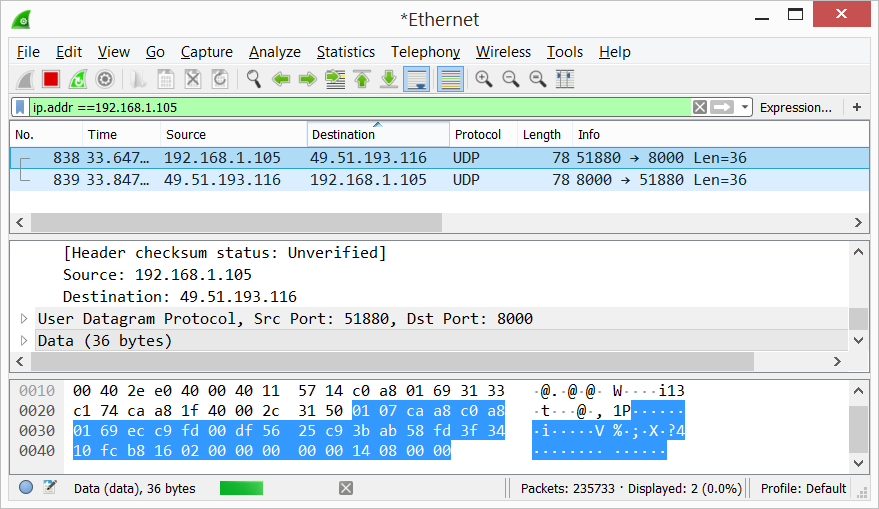
One thing I'd like to do is use my own servers to replace the Chinese servers that the camera communicates to. So I looked into the protocol that the camera utilises to communicate with the servers.

The camera sends a ping to the server every minute or so as a keep-alive, which means the camera keeps sending small packets of data to the server to inform it that the camera is still powered on and working.

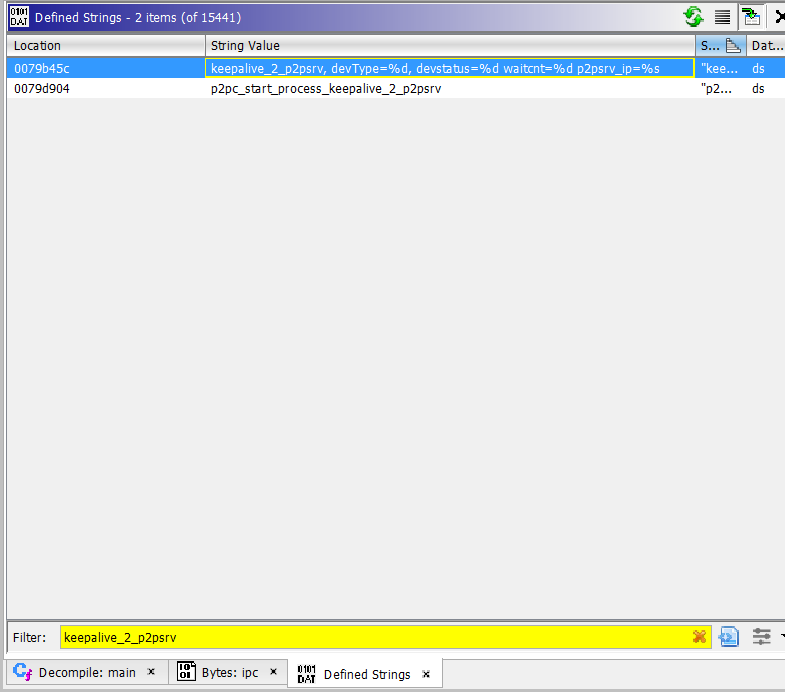
This is detectable by capturing traffic with WireShark but is also visible at terminal, since IPC prints the following information for every keep-alive packet it sends and receives:



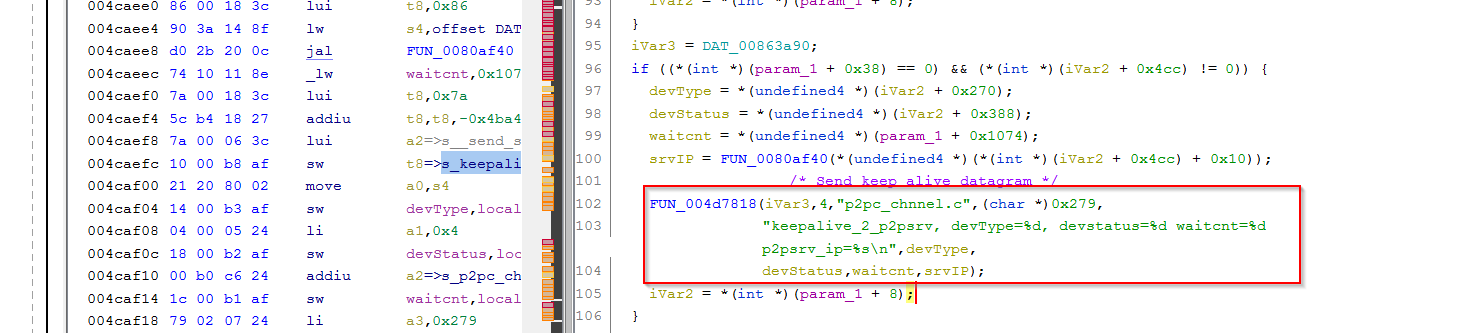
And the data exchanged by camera and server during this keep-alive session can be seen on WireShark, as shown here:



So, lets look where in the code IPC is sending this string to the terminal. Ghidra has a great resource of listing all strings found inside the program:



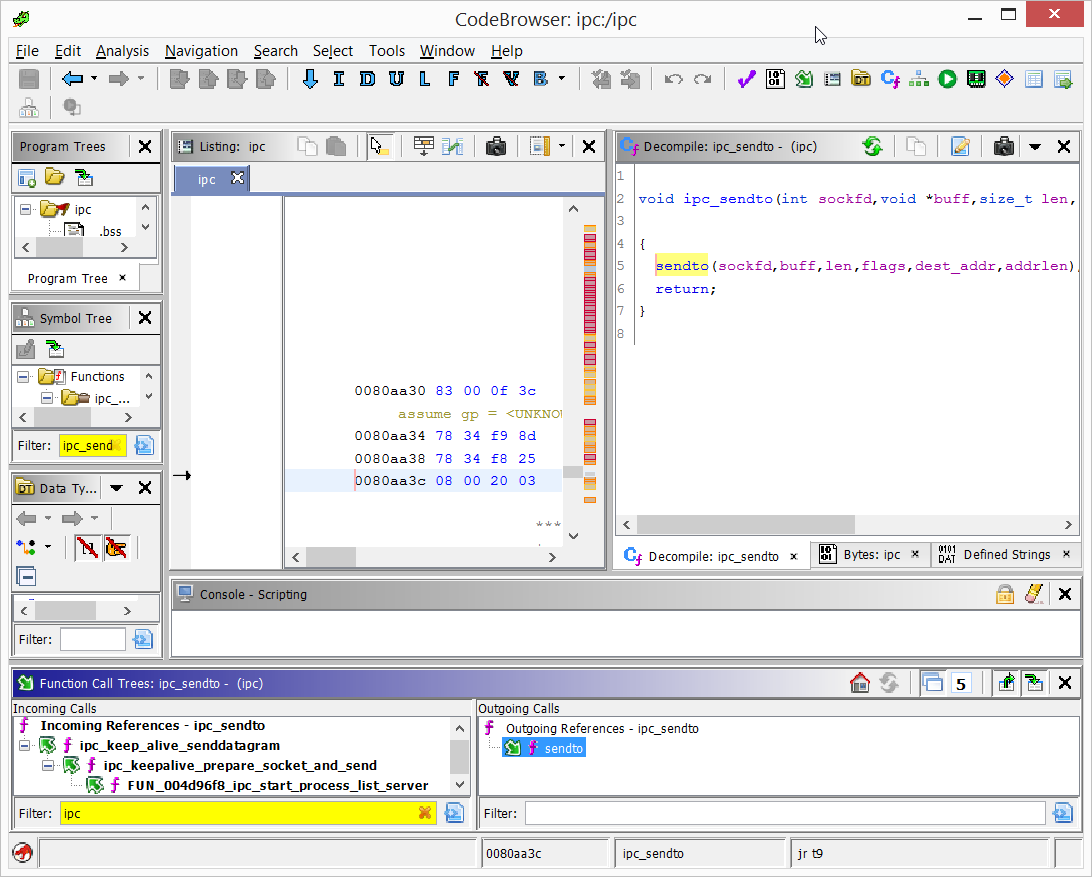
It also allows us to pin-point the exact piece of code that uses that string:



So, "keepalive 2 p2psrv, devType= % d, devstatus= % d waitcnt= % d p2psrv ip= % s" is supplied to a function that would most likely output it on the console (as we saw in the terminal screen shot at the beginning of this article). The keep-alive UDP datagram that IPC uses to convey this data is most likely mounted somewhere. Let's look into this further.

Looking for the Linux standard socket functions is an excellent method to figure out where the keep-alive UDP datagram is being sent. In this situation, the typical functions for sending UDP datagrams would be send() or sendto(). Ghidra is also excellent at displaying the call diagram of a series of functions.

We can see that there is a function wrapper (which I renamed to ipc sendto()) that is called by numerous functions inside the IPC programme by filtering the name "sendto" in Ghidra Symbol Tree ("Incoming calls" at the bottom left). We can easily find where the heart-beat code calls sendto() by extending each of these calls in the incoming calls tree and matching one of the sendto() calls to the identical function we located using the terminal string (above). Please notice that I renamed a lot of the functions throughout the code analysis process. That's how reverse engineering usually works: You begin with generic function names and gradually add comments and rename the functions as you learn more about what they perform.



This is how we came to establish that our camera was communicating with a china based server, owned by Shenzhen RUI speed cloud computing limited. This, alongside our root access to the operating system mounted on the SoC the camera came with are demonstrations enough to showcase how poorly secure budget IP Cameras are.

**Chapter 5**

**CONCLUSION AND SUGGESTIONS**

“

All in all, we were successfully able to bring to light the security flaws present in the currently available sea of budget IP cameras. We suggest the use of proven security principles like input validation, boundaries checking, access control, and authentication when creating a smart device. Encrypting communications is beneficial, but if the keys are easily obtained, it only delays rather than deters an attacker. Keys should be kept as secure as feasible, not stored in logs, and not established by an untrustworthy party.

When an attacker gains access to the device, for example, through unencrypted network binaries on the core System on Chip firmware, basic measures such as defence-in-depth and least privilege should be used so that they cannot do as much as if they had root capabilities.

To prevent bad firmware uploads, firmware should be encrypted at the very least. Many modern smart devices fail to puts both security and financial structures at risk, including this camera, and depend on their firmware to bind the customer in a monthly subscription model. This cloud-based business model is threatened by open source firmware (e.g. jailbreaking), as users might get an opportunity to take over their device’s credentials such as FTP (File Transfer Protocol) or their Emails.”