390R Final Writeup on the Wyze Cam V2

Ian Anderson & Nam Nyguen

May 23, 2023

1 Overview

The Wyze Cam V2 is a low cost security camera (MSRP $26) developed by Wyze Labs, Inc. As with most Wyze cameras, it is largely a licenced copy of an existing Xiaomi camera, in this case being based on the Xiaomi Xiaofang 1S.

Some notable features of this camera include:

- 1080p sensor with a 110 degree wide angle lens
- F2.0 aperture + 4 infrared LEDs, good for low light
- Speaker & microphone for two way communication, so it can be used as an intercom
- Accompanying mobile app to see live video feed, capture images & bitrate information

- microSD card slot for storage

Our goal with this project was to see if we could find a way to exploit security vulerabilities in this camera through the process of extracting and examining it's firmware.

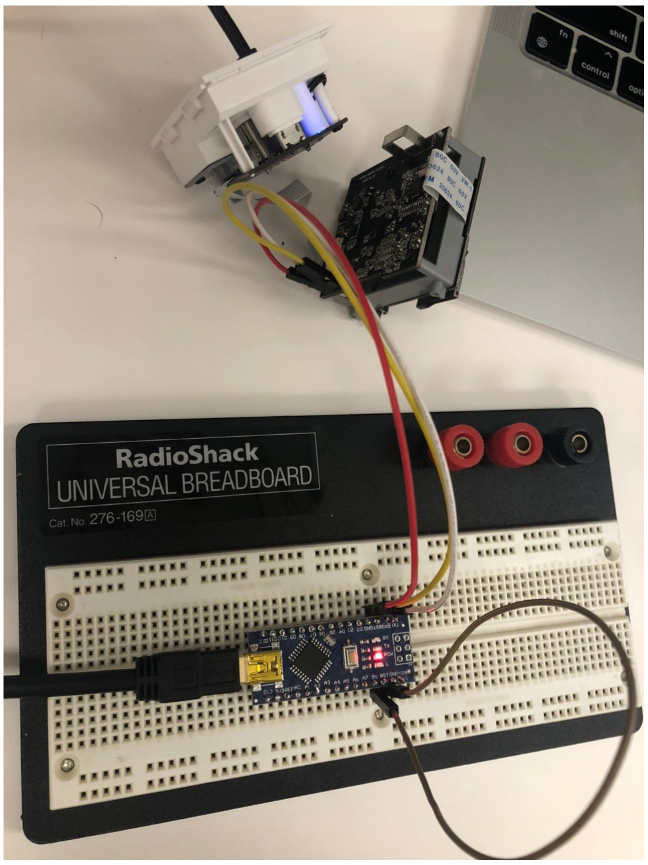
2 Technical Details

The Wyze Cam V2 is powered by the Ingenic T20 processor, an efficient SOC designed for video related IOT devices. Some of it's specs include:

- "XBurst" 1GHz single core CPU based on the MIPS32 architecture
- 128MB of DDR2 memory
- Hardware H.264 encoder supporting 1080p@30fps
- 600mw power draw

Something that stood out on the product info page was "Linux BSP, GCC tool chain, Glibc", confirming that this device most likely ran Linux.

3 Attempting to dump the firmware



We obtained two cameras on eBay, and after testing one of them to make sure it was fully working, went and disassembled it until we were able to take out the main board and the attached camera board. To gain shell access to the Linux system, we did some research and found that a hobbyist had figured out that three pins on the edge of the main corresponded to GND, RX and TX, and used an Arduino to gain serial access to the device. Using tools in the CICS Makerspace, we were able to replicate this and get to the login, but none of the passwords we found worked for the root account, so we were not able to directly get the firmware off the device itself. However, Wyze has versions of the device's firmware avaliable for download, though these are not the standard versions. We will examine the webcam firmware found on the Wyze website. (We later found a version of the RTSP firmware that was removed from the website some time ago, but this writeup will be focued on the webcam version.)

4 Basic examination

To start off, one of the first things we tried was using the `strings` command to see if we could find any useful data, as trying to read the binary just with `cat` would just result in a bunch of garbage being outputted to the screen, and `xxd` would take forever to go through with a binary of this size (11MB). To reduce the chance of simply getting a bunch of random valid characters in a row, we used the `-n16` flag to only print out "words" of 16 characters or greater, which resulted in this output:

```
Fn.record_auto_list
AppVer=4.15.2.82
wpa_supplicant -D nl80211 -iwlan0 -c/system/bin/wpa.conf -B &
udhcpc -i wlan0 -p /var/run/udhcpc.pid -b &
```

```
+m=DGEhostapd_cli
hostapd_wpa2.conf
'_ _ _$_$_"_"_&_&_!_!_%_%_#_#_'_'
'? ? ?$?$?"?"?&?&?!?!?%?%?#?#?'?'
'_ _ _$_$_"_"_&_&_!_!_%_%_#_#_'_'
'? ? ?$?$?"?"?&?&?!?!?%?%?#?#?'?'
mount -o nolock,rw 10.0.0.167:/home/xuxuequan/Ingenicwork/sharenfs /mnt
restart_wlan0.sh
az#=f02`F##f22af#3f1
:*Fp"`D'"Fr"ad'2Fq
az%=fP2`F%#fR2af%3fQ
zISA_VERSION=5.6.1.32
root:x:0:0:root:/:/bin/sh
x:root:rJOFHsG0ZbyZo:10933:0:99999:7:::
webrtc_profile.ini
wpa_supplicant.conf
ctrl_interface=/var/run/wpa_supplicant
model=isa.camera.isc5c1
mac=34:CE:00:E3:FD:D7
key=V2RX3WeYFLdWeEZc
&PsD7p5STEDiLK4DV
miio_client_helper_nomqtt.sh
videobuf2-vmalloc.ko
WH@XXH@HPDLFBDTZTPHRQRZVNAAAXBYUI^UF^A
+8$4,<"2*9%5-=#3
}Rlibsysutils.so
dongle_network_add_failed.wav
dongle_network_add_success.wav
E6rdongle_network_start.wav
Jdongle_sensor_delete.wav
root:rJOFHsG0ZbyZo:10933:0:99999:7:::
7root:rJOFHsG0ZbyZo:10933:0:99999:7:::
root:rJOFHsG0ZbyZo:10933:0:99999:7:::
root:rJOFHsG0ZbyZo:10933:0:99999:7:::
=root:rJOFHsG0ZbyZo:10933:0:99999:7:::
iroot:rJOFHsG0ZbyZo:10933:0:99999:7:::
root:rJOFHsG0ZbyZo:10933:0:99999:7:::
Kroot:rJOFHsG0ZbyZo:10933:0:99999:7:::
root:rJOFHsG0ZbyZo:10933:0:99999:7:::
root:rJOFHsG0ZbyZo:10933:0:99999:7:::
K!uvc_f22.config
root:rJOFHsG0ZbyZo:10933:0:99999:7:::
root:rJOFHsG0ZbyZo:10933:0:99999:7:::
G8uvc_jxf22.config
uvc_jxf23.config
[Qbnjp~iQr*V=s&:1$Lv
@VYSGFQ]KJ^UCRNYMBFQELZ^ITRVADBFNX\JFHTRJPD\B@XL
+8$4,<"2*9%5-=#3
root:rJOFHsG0ZbyZo:10933:0:99999:7:::
```

```
croot:rJ0FHsG0ZbyZo:10933:0:99999:7:::
libaudioProcess.so
```

While a few of these were still garbage, most of this was useful information that was telling about the development of this system. For example, `/home/xuxuequan/Ingenicwork/sharenfs` seems to be an NFS share used by the developers of this firmware at Ingenic. Most useful though are the shadow hashes, such as `x:root:rJ0FHsG0ZbyZo:10933:0:99999:7:::`, since these can be used to potentially crack the password for the root account. To get a better idea of where these strings are, we need to explore the file system

5 File system exploration

To extract the file system from the firmware, we can use the tool binwalk. Running the tool without any additional flags gives us some basic information

```
DECIMAL       HEXADECIMAL     DESCRIPTION
--------------------------------------------------------------------------------
0             0x0             uImage header, header size: 64 bytes, header CRC: 0x413EFAFE,
created: 2020-03-21 10:10:07, image size: 11075584 bytes, Data Address: 0x0, Entry Point:
0x0, data CRC: 0xC181F3AB, OS: Linux, CPU: MIPS, image type: Firmware Image, compression
type: none, image name: "jz_fw"
64            0x40            uImage header, header size: 64 bytes, header CRC: 0x7771D66A,
created: 2020-03-20 15:44:39, image size: 1816781 bytes, Data Address: 0x80010000, Entry
Point: 0x803E9230, data CRC: 0xBDC20C2D, OS: Linux, CPU: MIPS, image type: OS Kernel Image,
compression type: lzma, image name: "Linux-3.10.14"
128           0x80            LZMA compressed data, properties: 0x5D, dictionary size:
67108864 bytes, uncompressed size: -1 bytes
2097216       0x200040        Squashfs filesystem, little endian, version 4.0,
compression:xz, size: 3353204 bytes, 407 inodes, blocksize: 131072 bytes,
created:2019-05-21 17:22:45
5570624       0x550040        Squashfs filesystem, little endian, version 4.0,
compression:xz, size: 583802 bytes, 13 inodes, blocksize: 131072 bytes,
created: 2020-02-23 16:07:18
6225984       0x5F0040        JFFS2 filesystem, little endian
8072064       0x7B2B80        JFFS2 filesystem, little endian
8082472       0x7B5428        JFFS2 filesystem, little endian
8162316       0x7C8C0C        Zlib compressed data, compressed
8165504       0x7C9880        Zlib compressed data, compressed
8168320       0x7CA380        Zlib compressed data, compressed
8170872       0x7CAD78        JFFS2 filesystem, little endian
...
11068972      0xA8E62C        JFFS2 filesystem, little endian
11069100      0xA8E6AC        Zlib compressed data, compressed
11070652      0xA8ECBC        JFFS2 filesystem, little endian
11072816      0xA8F530        Zlib compressed data, compressed
```

The majority of the binary is made up of the JFFS2 file system used by the Linux install (shown to be on kernel 3.10.14), with much of it being compressed. Installing Jefferson, sasquatch and squashfs-tools allows us to run binwalk with the `-e` flag and extract the root Linux filesystem. Looking through the squashfs file system for custom files, one that immediately stuck out to me was `/etc/init.d/rcS`. This is a shell script that sets up the system:

```
#!/bin/sh

# Set mdev
echo /sbin/mdev > /proc/sys/kernel/hotplug
/sbin/mdev -s && echo "mdev is ok......"

# create console and null node for nfsroot
#mknod -m 600 /dev/console c 5 1
#mknod -m 666 /dev/null c 1 3

# Set Global Environment
export PATH=/bin:/sbin:/usr/bin:/usr/sbin
export PATH=/system/bin:$PATH
export LD_LIBRARY_PATH=/system/lib
export LD_LIBRARY_PATH=/thirdlib:$LD_LIBRARY_PATH

# networking
ifconfig lo up
#ifconfig eth0 192.168.1.80

# Start telnet daemon
telnetd &

# Set the system time from the hardware clock
#hwclock -s

#set the GPIO PC13 to high, make the USB Disk can be use
cd /sys/class/gpio
[Note: the following sentences in brackets were originally in Chinese, I used Google Transla
echo 77 > export       #[Application]GPIO
cd gpio77
echo out > direction   #[set to output mode]
echo 0 > active_low    #value[It is 0, which means low level] value[It is 1, indicating high
echo 1 > value         #[Setting Level (Output Mode)]

# Mount driver partition
mount -t squashfs /dev/mtdblock3 /driver

# Mount system partition
mount -t jffs2 /dev/mtdblock4 /system

# Mount backup partition
#mount -t jffs2 /dev/mtdblock5 /backupk

# Mount backup partition
#mount -t jffs2 /dev/mtdblock6 /backupd

# Mount backup partition
mount -t jffs2 /dev/mtdblock7 /backupa
```

```
# Mount configs partition
mount -t jffs2 /dev/mtdblock8 /configs

# Mount params partition
mount -t jffs2 /dev/mtdblock9 /params

# Format system patition if it is invalid
if [ ! -f /system/.system ]; then
    echo "Format system partition..."
    umount -f /system
    flash_eraseall /dev/mtd4
    mount -t jffs2 /dev/mtdblock4 /system
    cd /system
    mkdir -p bin init etc/sensor lib/firmware lib/modules
    echo "#!/bin/sh" > init/app_init.sh
    chmod 755 init/app_init.sh
    touch .system
    cd /
    echo "Done"
fi

# Run init script
if [ -f /system/init/app_init.sh ]; then
    /system/init/app_init.sh &
fi
```

This shows how the install is broken up into a number of different file systems. Looking at the main JFFS2 file system is far more interesting. Most of it is made up of shell scripts and custom binaries. Starting off with `/init/app_init.sh`:

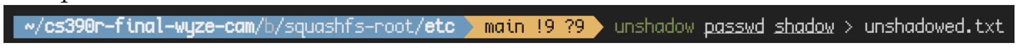
```
#!/bin/sh

cd /system
if [ -f /system/.upgrade ]; then
    cd /backupa
    echo "init upgrading!!!!!!!!!!!"
    ./upgrade.sh
    rm /system/.upgrade
fi

##insmod /lib/modules/tx-isp.ko isp_clk=100000000
##insmod /lib/modules/sensor_imx323.ko
##insmod /lib/modules/sensor_jxf22.ko
##insmod /lib/modules/sensor_ps5230.ko
##insmod /lib/modules/exfat.ko
##insmod /lib/modules/sample_motor.ko
##insmod /lib/modules/audio.ko
##insmod /lib/modules/sinfo.ko
##insmod /lib/modules/8189es.ko
```

```
insmod /driver/tx-isp.ko isp_clk=100000000
##insmod /driver/sensor_imx323.ko
#insmod /driver/sensor_jxf23.ko
##insmod /driver/sensor_ps5230.ko
insmod /driver/exfat.ko
insmod /driver/sample_motor.ko
insmod /system/audio.ko
#spk_gpio=-1 sign_mode=0
insmod /driver/sinfo.ko
##insmod /driver/8189es.ko
insmod /driver/sample_pwm_core.ko
insmod /driver/sample_pwm_hal.ko
#insmod /driver/rtl8189ftv.ko
insmod /system/libcomposite.ko
insmod /system/videobuf2-vmalloc.ko
insmod /system/usbcamera.ko

sh /system/bin/led.sh &
/system/bin/getSensorType
/system/ucamera &

#wpa_supplicant -Dwext -i wlan0 -c /system/etc/wpa_supplicant.conf -B
#udhcpc -i wlan0 -s /system/etc/udhcpc.script -q

ifconfig eth0 up
#udhcpc -i eth0 -s /system/etc/udhcpc.script -q
ifconfig eth0 193.169.4.222
route add default gw 193.169.4.1

mount -t nfs -o nolock 193.169.4.2:/home_b/nfsroot/ywu/ /mnt
# open ircut
#cp /system/bin/setir /tmp/
#config ip address

echo 63 > /sys/class/gpio/export
echo "out" > /sys/class/gpio/gpio63/direction
echo 1 > /sys/class/gpio/gpio63/value
#/system/bin/carrier-server --st=imx322
#/system/bin/singleBoadTest
if [ -f /system/bin/.debug ]; then
    echo "root:rJ0FHsG0ZbyZo:10933:0:99999:7:::" > /system/etc/shadow
#    /system/bin/iCamera
else
    echo "root:x:10933:0:99999:7:::" > /system/etc/shadow
#    /system/bin/iCamera >/dev/null 2>&1
fi
```

Something interesting that I noticed was all the commented lines. This makes sense when you consider the fact that, with the Ingenic T20 being used in a number of security cameras, that each would have different features that would require different kernel modules. Also, it is interesting that if the `.debug` file exists, then the shadow file is different.

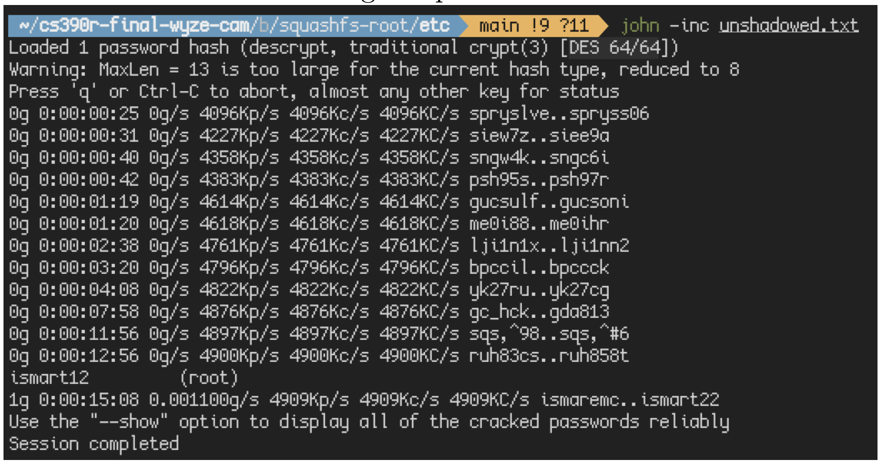
6 Cracking the root password

With the shadow and passwd files, we can use John the Ripper to crack the root password. Since the hash used is DES, this was very easy, even when using pure brute force. First, we combine the shadow and passwd files:



```
~/cs390r-final-wyze-cam/b/squashfs-root/etc  main !9 ?9  unshadow passwd shadow > unshadowed.txt
```

Then we brute force cracking the password:

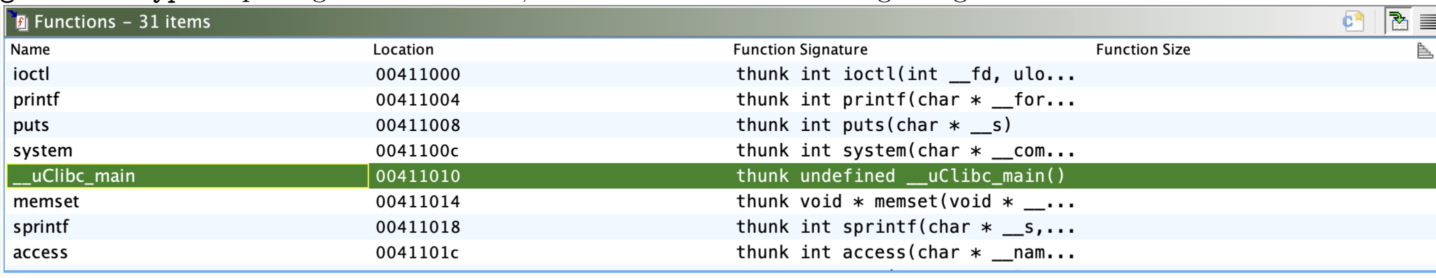


```
~/cs390r-final-wyze-cam/b/squashfs-root/etc  main !9 ?11  john -inc unshadowed.txt
Loaded 1 password hash (descrypt, traditional crypt(3) [DES 64/64])
Warning: MaxLen = 13 is too large for the current hash type, reduced to 8
Press 'q' or Ctrl-C to abort, almost any other key for status
0g 0:00:00:25 0g/s 4096Kp/s 4096Kc/s 4096KC/s spryslve..spryss06
0g 0:00:00:31 0g/s 4227Kp/s 4227Kc/s 4227KC/s siew7z..siee9a
0g 0:00:00:40 0g/s 4358Kp/s 4358Kc/s 4358KC/s sngw4k..sngc6i
0g 0:00:00:42 0g/s 4383Kp/s 4383Kc/s 4383KC/s psh95s..psh97r
0g 0:00:01:19 0g/s 4614Kp/s 4614Kc/s 4614KC/s gucsulf..gucsoni
0g 0:00:01:20 0g/s 4618Kp/s 4618Kc/s 4618KC/s me0i88..me0ihr
0g 0:00:02:38 0g/s 4761Kp/s 4761Kc/s 4761KC/s lji1n1x..lji1nn2
0g 0:00:03:20 0g/s 4796Kp/s 4796Kc/s 4796KC/s bpccil..bpccck
0g 0:00:04:08 0g/s 4822Kp/s 4822Kc/s 4822KC/s yk27ru..yk27cg
0g 0:00:07:58 0g/s 4876Kp/s 4876Kc/s 4876KC/s gc_hck..gda813
0g 0:00:11:56 0g/s 4897Kp/s 4897Kc/s 4897KC/s sqs,^98..sqs,^#6
0g 0:00:12:56 0g/s 4900Kp/s 4900Kc/s 4900KC/s ruh83cs..ruh858t
ismart12         (root)
1g 0:00:15:08 0.001100g/s 4909Kp/s 4909Kc/s 4909KC/s ismaremc..ismart22
Use the "--show" option to display all of the cracked passwords reliably
Session completed
```

So, the password ends up being `ismart12`.

7 Custom Binary analysis with Ghidra

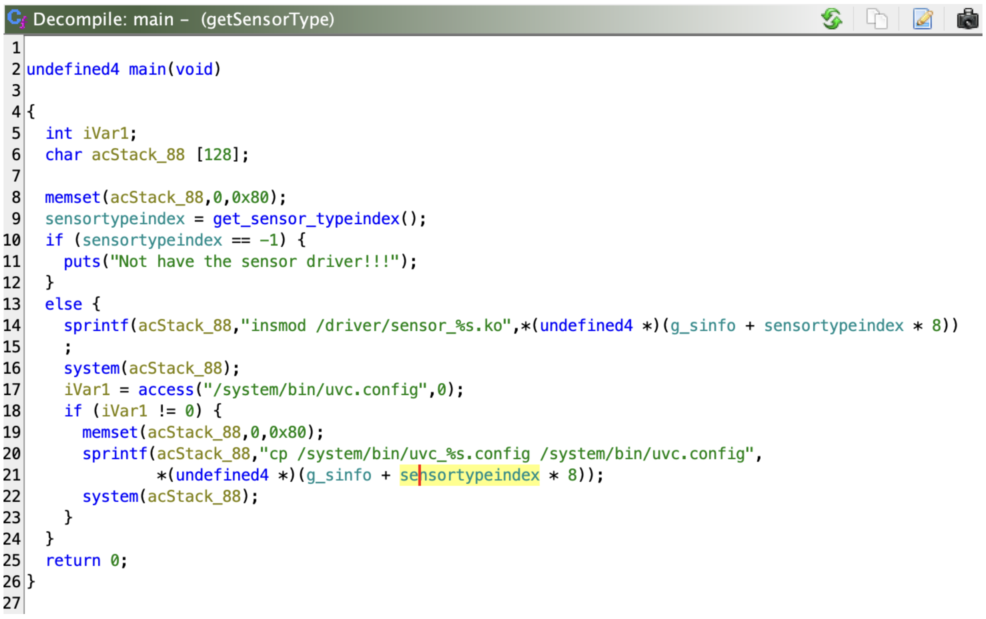
One more thing to look at was some of the custom binaries. Many of them are pretty simple, such as `getSensorType`. Opening it with Ghidra, we can see a few interesting things.



Functions - 31 items

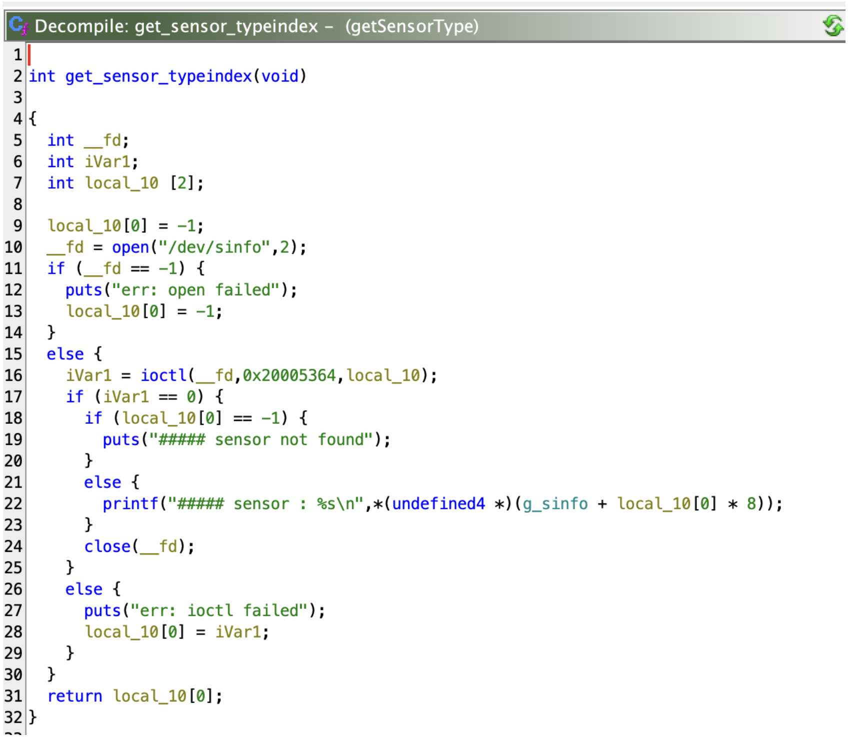
Name	Location	Function Signature	Function Size
ioctl	00411000	thunk int ioctl(int __fd, ulo...	
printf	00411004	thunk int printf(char * __for...	
puts	00411008	thunk int puts(char * __s)	
system	0041100c	thunk int system(char * __com...	
__uClibc_main	00411010	thunk undefined __uClibc_main()	
memset	00411014	thunk void * memset(void * __...	
sprintf	00411018	thunk int sprintf(char * __s,...	
access	0041101c	thunk int access(char * __nam...	

First of all, all of these binaries were compiled with uClibc, a C library meant for embedded systems such as this one.



```

undefined4 main(void)

{
  int iVar1;
  char acStack_88 [128];

  memset(acStack_88,0,0x80);
  sensortypeindex = get_sensor_typeindex();
  if (sensortypeindex == -1) {
    puts("Not have the sensor driver!!!");
  }
  else {
    sprintf(acStack_88,"insmod /driver/sensor_%s.ko",*(undefined4 *)(g_sinfo + sensortypeindex * 8))
    ;
    system(acStack_88);
    iVar1 = access("/system/bin/uvc.config",0);
    if (iVar1 != 0) {
      memset(acStack_88,0,0x80);
      sprintf(acStack_88,"cp /system/bin/uvc_%s.config /system/bin/uvc.config",
              *(undefined4 *)(g_sinfo + sensortypeindex * 8));
      system(acStack_88);
    }
  }
  return 0;
}

```

Here is the main function of this program. As you can see, it opens a config file before trying to get the sensor type.



```

int get_sensor_typeindex(void)

{
  int __fd;
  int iVar1;
  int local_10 [2];

  local_10[0] = -1;
  __fd = open("/dev/sinfo",2);
  if (__fd == -1) {
    puts("err: open failed");
    local_10[0] = -1;
  }
  else {
    iVar1 = ioctl(__fd,0x20005364,local_10);
    if (iVar1 == 0) {
      if (local_10[0] == -1) {
        puts("##### sensor not found");
      }
      else {
        printf("##### sensor : %s\n",*(undefined4 *)(g_sinfo + local_10[0] * 8));
      }
      close(__fd);
    }
    else {
      puts("err: ioctl failed");
      local_10[0] = iVar1;
    }
  }
  return local_10[0];
}
```

To get the sensor type, it simply opens `/dev/sinfo` and checks if a certain IO device exists. There are other binaries like this in the image that do similarly simple functions

8 Challenges

The biggest challenge for us was getting into the file system. We initially thought this firmware would be akin to a ROM dump, and would be able to be directly analyzed in Ghidra. However, closer analysis revealed that it is more like a disk image, with its own file system that needed to be separately extracted. Additionally, getting the actual cameras took some time, as they had to be purchased by us.

9 Further goals

If we had more time, we would focus on analyzing all the custom binaries, looking for vulerabilities, as well as flashing this webcam firmware to one of the devices and running potential exploits on that. Since there are many custom binaries here, it is likely that at least one has potential exploits, and probably quite a few of them.