ENTRY // Xiongmai Security Research #2 // Software Overview and Audit [ — ] [ × ]

**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 theXMEye 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.

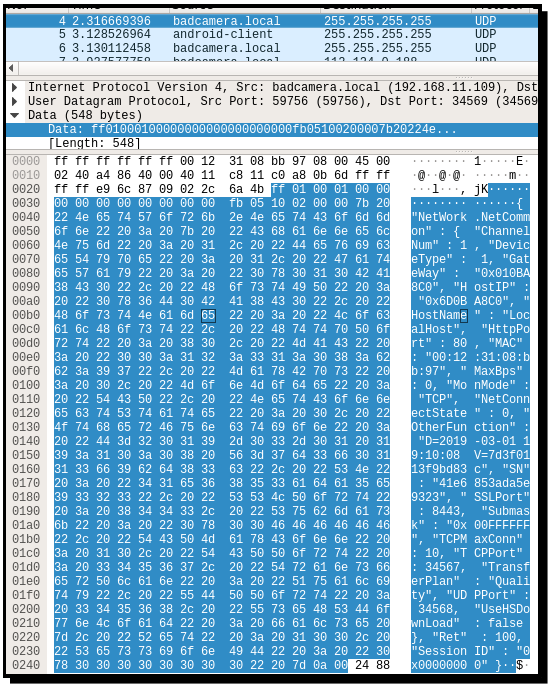
```
 3 2.221616359   android-client      255.255.255.255   UDP    64 synapsis-edge(5008) → 34569 Len=20
 4 2.316669396   badcamera.local     255.255.255.255   UDP   592 59756 → 34569 Len=548
 5 3.128526964   android-client      255.255.255.255   UDP    64 synapsis-edge(5008) → 34569 Len=20
 6 3.130112458   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.

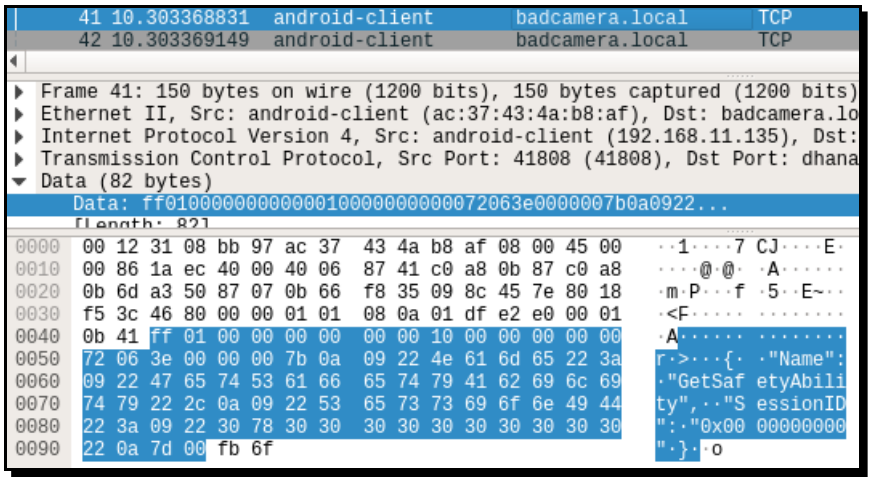
Taking a look at the first packet, we are greeted with a 20 byte sequence, a single byte 0xFF, padded by thirteen 0x00, then 0xFA05, and lastly four 0x00. We will call this the DBP (Discovery Broadcast Packet).

```
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)

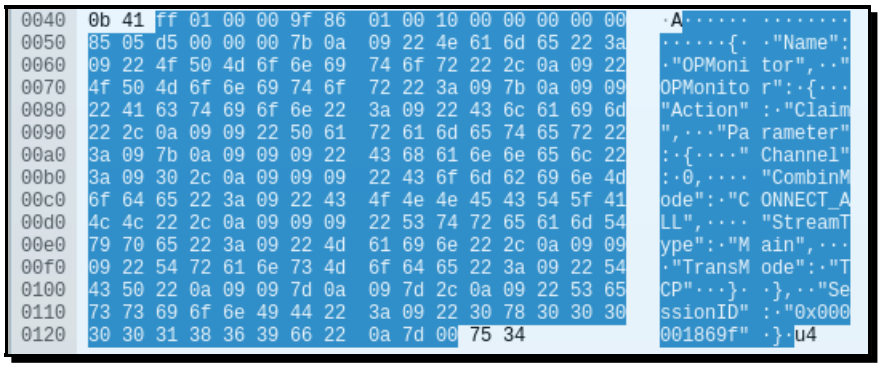



It does this little dance to share the UDPPort and the TCPPort, both of which are most likely hard-coded in, so there isn't much of a reason for it. We also know that "Ret : 100" means the command succeeded!




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.

```c
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.

```c
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.

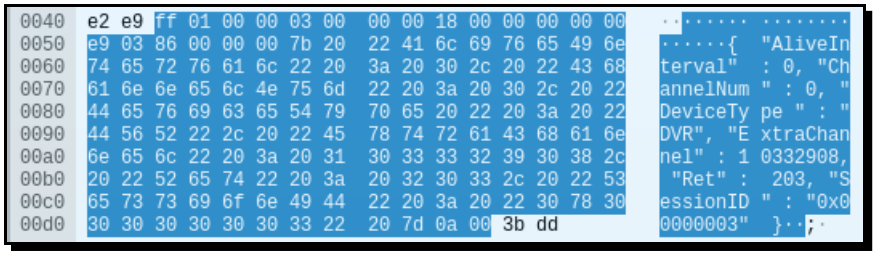
```
0040  e2 e0 ff 01 00 00 00 00  00 00 10 00 00 00 00 00   ······· ·····
0050  73 06 64 00 00 00 7b 20  22 4e 61 6d 65 22 20 3a   s·d···{  "Name" :
0060  20 22 47 65 74 53 61 66  65 74 79 41 62 69 6c 69    "GetSaf etyAbili
0070  74 79 22 2c 20 22 52 65  74 22 20 3a 20 31 30 33   ty", "Re t" : 103
0080  2c 20 22 53 65 73 73 69  6f 6e 49 44 22 20 3a 20   , "Sessi onID" :
0090  22 30 78 30 30 30 30 30  30 30 30 30 30 22 2c 20   "0x00000 00000",
00a0  22 61 75 74 68 6f 72 69  7a 65 53 74 61 74 22 20   "authori zeStat"
00b0  3a 20 6e 75 6c 6c 20 7d  0a 00 2d 83               : null } ··-·
```

```
0040  e2 e1 ff 01 00 00 9f 86  01 00 11 00 00 00 00 00   ······· ·····
0050  86 05 43 00 00 00 7b 20  22 4e 61 6d 65 22 20 3a   ··C···{  "Name" :
0060  20 22 4f 50 4d 6f 6e 69  74 6f 72 22 2c 20 22 52    "OPMoni tor", "R
0070  65 74 22 20 3a 20 31 30  33 2c 20 22 53 65 73 73   et" : 10 3, "Sess
0080  69 6f 6e 49 44 22 20 3a  20 22 30 78 30 30 30 31   ionID" :  "0x0001
0090  38 36 39 46 22 20 7d 0a  00 6c 82                   869F" }· ·l·
```

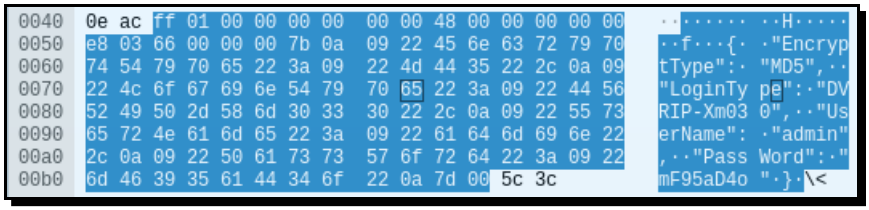
Now we finally start to get to the meat. The client attempts to login to the device, but I have changed the default password to "password", whatever this password is, it's most likely the blank password. The reply afterwards should be a "failure", we can see that the next login attempt has a different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

```
0040  0b 44 ff 01 00 00 00 00  00 00 18 00 00 00 00 00   ·D····· ·····
0050  e8 03 66 00 00 00 7b 0a  09 22 45 6e 63 72 79 70   ··f···{· ·"Encryp
0060  74 54 79 70 65 22 3a 09  22 4d 44 35 22 2c 0a 09   tType":· "MD5",··
0070  22 4c 6f 67 69 6e 54 79  70 65 22 3a 09 22 44 56   "LoginTy pe":·"DV
0080  52 49 50 2d 58 6d 30 33  30 22 2c 0a 09 22 55 73   RIP-Xm03 0",··"Us
0090  65 72 4e 61 6d 65 22 3a  09 22 61 64 6d 69 6e 22   erName": ·"admin"
00a0  2c 0a 09 22 50 61 73 73  57 6f 72 64 22 3a 09 22   ,··"Pass Word":·"
00b0  74 6c 4a 77 70 62 6f 36  22 0a 7d 00 e4 c3         tlJwpbo6 "·}···
```

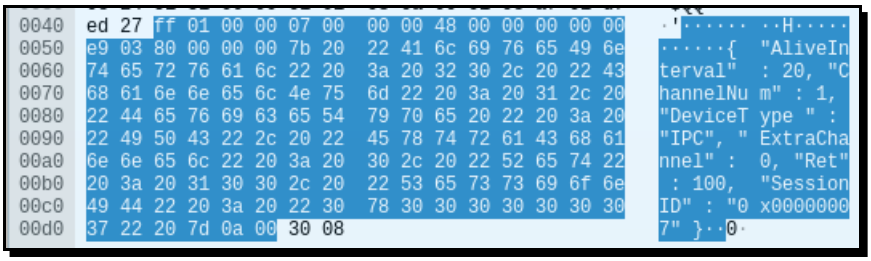
Blank Password




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.

---

TAGGED SKILLS:

Security Research | Exploitation | C

+--------------------------------------------------+
© 2026 Ian Rash - Built with Crystal & Made into a PDF
+--------------------------------------------------+