Wyze Camera Fall 2023

Michael Edoigiawerie, Varsha Jacob, Joshua Muehring, Spencer Redelman, Deniz Timurturkan

October 2023

Embedded Systems Cyber Security
Georgia Institute of Technology
Atlanta, Georgia

1 Abstract

This paper covers details about the Wyze Camera and the Georgia Tech Embedded System Cyber Security's research into the camera as of Fall 2023. The goal of the research project is to exploit the Wyze IP Camera, which can be done by manually reverse engineering the camera's code for receiving and transmitting information. The overall goal of the project is to reverse engineer the Wyze Camera's RF protocols in order to exploit it as a proof of concept. The team will then try to automatically get these same results through a process called fuzzing as a proof of concept. The results of these two processes will then be compared to see if fuzzing is a viable alternative for manual reverse engineering RF devices.

2 Introduction

The Wyze Camera is an IoT (Internet of Things) device that allows users to view remote locations through the Wyze App or the Wyze Web View. Users can set up the camera in their homes in order to have a feed of their homes, pets, or children while they are away. After setting up the camera, authenticating and connecting the camera with their personal devices, users are able to stream the camera's view on their personal devices.

However, both cameras and IoT devices in general can bring many risks that are not yet fully taken seriously by the companies producing these devices and the users. The Wyze Camera itself had vulnerabilities that allowed users to connect to a device without authentication, as well as bugs that showed users other camera feeds on the Wyze Web View for a short period of time. [1]

Furthermore, IoT devices in general bring along with them many security risks that must be taken seriously. IoT devices can pose security risks from the device itself, from the transmission protocol used to communicate between devices, and in the application or website that connects to the device [2]. Further adding to the impact of this is the fact that many users of IoT devices are not always knowledgeable or willing enough to take extra measures to mitigate their risks.

For example, it is often recommended that IoT devices use their own network or network segment, but many users just add their devices to their home networks. Though this may be easier than setting up a new network or sub-network, it may also mean that the IoT devices are now running on the same network that the user's other devices are running on. This means that if the IoT device gets hacked, it could allow the hacker to access sensitive data stored on the network or on other devices on the network [3]. On the other hand, another device on the network could be able to hack into the IoT device. This is why it is vital that IoT devices have strong security, since many users themselves do not have the knowledge to mitigate their own security risks. Thus, it is important to find potential vulnerabilities in IoT, as well as a way to automatically find vulnerabilities, in order to better increase the security of the devices themselves and decrease the burden of security on the users of the devices, who may not have the knowledge, time, or energy to take the necessary steps to protect their data.

3 Device Description

The Wyze Camera works on data from two sensors, the contact sensor and the motion sensor [4]. These two peripheral sensors send their data to the sensor bridge using wireless communication. The sensor bridge then communicates with the camera's processors through USB. The camera's processors finally use wifi in order to send the data to the Wyze server and Wyze user application, which is available both as a mobile app and as an online web service. This communication can be seen in the image below:

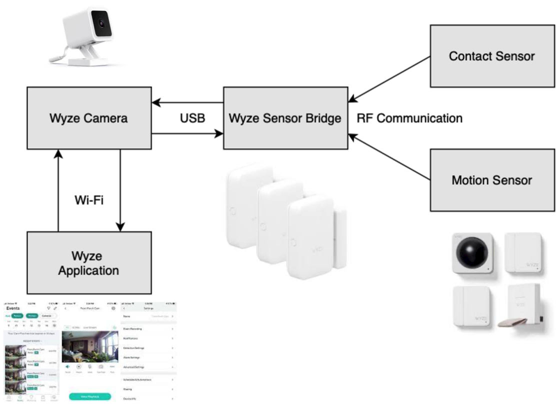

Figure 1: A diagram detailing the Wyze camera's components and the communication between them.

The contact and motion sensors use wireless communication to communicate with the sensor bridge. It uses the TI CC1310 microcontroller, which supports a variety of data rates and modulation schemes.

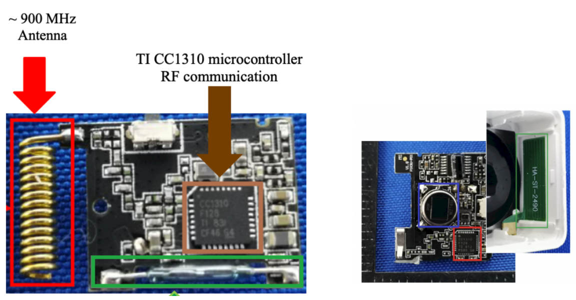

Figure 2: The contact (left) and motion (right) sensors on the Wyze IP Camera

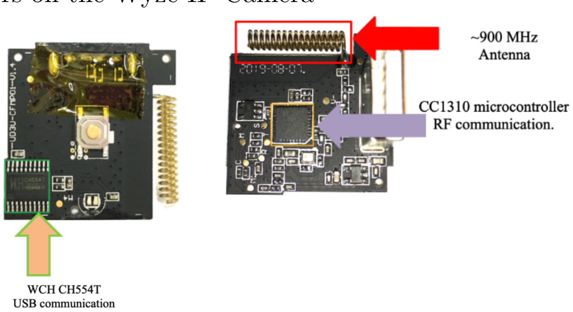

Figure 3: Back (left) and front (right) of the sensor bridge.

The TI CC1310 is a microcontroller composed of two processors, as well as peripheral controllers. The first processor, the ARM Cortex M3, is the main processor. This processor is part of the the system-side and runs the user application. This processor also operates on the information from the TX and RX packets. The second processor, the ARM Cortex M0, is part of the radio-side and receives commands from the M3 processor. This processor creates packets to send to the M3 processor. Each processor is a separate system, but they work together and communicate with each other.

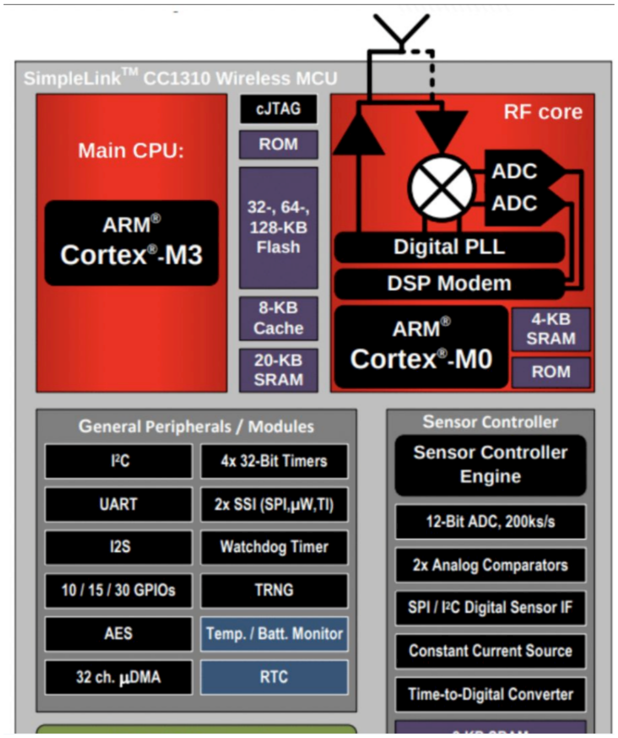

Figure 4: A diagram of the TI CC1310, its two processors, and peripheral units.

The hardware of the Wyze IP camera includes the Wyze Camera PCB, which runs embedded Linux. It also contains a wifi board for communication between the camera and the user application, as well as an open serial port with root permissions. It also includes an SD card slot to save footage.

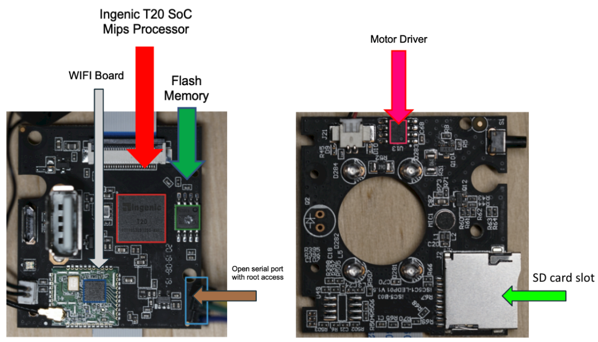

Figure 5: An image of the hardware of the Wyze IP Camera daugherboard

4 Existing Vulnerabilities

Previous vulnerabilities have been found in the Wyze Camera by various research teams. Some of these include the ability to bypass authentication and buffer overflows, among others [5]. The Wyze Camera needs a username and password to log into the device. The camera then sends a verification code to the user's device to connect the two. However, one bug allows for the authentication code to never be stored in the camera's memory, keeping the original NULL value instead. Then, by passing in the NULL value as the code, the camera will be linked to the device without actually getting the correct authentication code. Additionally, though the camera requires a password to authenticate, the password is still crackable, especially with a weak password, which many users may have.
Another vulnerability that has been found is a buffer overflow during the authentication process in the IOCtl (Input/Output Control). When the device gets an input, the input includes the size of the payload. However, the camera does not compare this value to the size of its destination buffer, which means that a large payload size will overflow the buffer and write outside of its allocated space.[5]

5 Analysis

During our research, we were focused on reverse engineering the camera's TX and RX packet structure, as well as reverse engineering the packet processing code in order to see how the payloads are structured, so that we are able to spoof a packet. The goal of our research is to be able to exploit the camera as a proof of concept by spoofing a packet, and the next steps of the research include attempting to use fuzzing to find vulnerabilities automatically.

Analysis of TX Protocol:
Packets from the Wyze camera are composed of a Preamble, Header, Syncword, a payload, and an optional CRC according to the TI CC13x0 advanced packet structure detailed in Figure 6. In order to prepare for packet transmission, the radio must be set up to use proprietary mode with the CMD_PROP_RADIO_DIV_SETUP command. To transmit a packet using advanced modes the TI CC13x0 utilizes the CMD_PROP_TX_ADV command. [4]

1 bit to 32 bytes or repetition	8 to 32 bits	0 to 32 bits	0 to 8 bytes	Arbitrary	0 or 16 bits (0 to 32 bits)
Preamble	Sync word	Header	Address	Payload	CRC

Figure 6: From the TI technical manual, an image of the available fields in the data packet

Importing the contents of the Wyze Camera SRAM into GHRIDA data types are identifies and usages of each type are determined. Researching previous semester's work the command structure rfc_CMD_PROP_TX_ADV_s was found to contain the important information regarding this TX command and the packet(s) being sent. Our goal for the rest of the semester was to investigate how/where this information is populated and how to create a packet in the same manner as to spoof an arbitrary packet as well as reverse engineer the exact functionality implemented by the creators of the Wyze Camera using the de-compiled instructions in GHIDRA as well as the specifications within the TI-CC13x0 manual.

Offset	Length	Mnemonic	DataType	Name
0x0	0x2	uint16_t	uint16_t	commandNo
0x2	0x2	uint16_t	uint16_t	status
0x4	0x4	rfc_radioOp_t *	rfc_radioOp_t *	pNextOp
0x8	0x4	ratmr_t	ratmr_t	startTime
0xc	0x1	_struct_147	_struct_147	startTrigger
0xd	0x1	_struct_148	_struct_148	condition
0xe	0x1	_struct_149	_struct_149	pktConf
0xf	0x1	uint8_t	uint8_t	numHdrBits
0x10	0x2	uint16_t	uint16_t	pktLen
0x12	0x1	_struct_150	_struct_150	startConf
0x13	0x1	_struct_151	_struct_151	preTrigger
0x14	0x4	ratmr_t	ratmr_t	preTime
0x18	0x4	uint32_t	uint32_t	syncWord
0x1c	0x4	uint8_t *	uint8_t *	pPkt

Figure 7: rfc_CMD_PROP_TX_ADV_s Command Structure

Notable Fields:

pPkt	uint8_t* pointer to packet to be transmitted, *if pktLen == 0* then pPkt points to a transmit queue instead of an individual packet (transmitting all data in the queue until empty, raising a TX_ENTRY_DONE interrupt when done)
preTrigger	struct containing Trigger information used to transition from transmission of preamble & syncWord to transmission packet (Preamble & syncWord repeated until preTrigger == TRIG_NOW)
syncWord	uint32_t word used to identify start of a packet in a rf connection
pktLen	uint16_t length of packet to be sent, 0 if a transmission queue is to be used
numHdrBits	uint8_t length of the header
pktConf	struct of bFsOff, bUseCrc, bCrcIncSw, bCrcIncHdr containing information regarding a Cyclic Redundancy Check (Checksum) of the packet to be sent
startConf	byte struct detailing if an external trigger or preTrigger is to be used, rising/falling edge information, and the input event used to capture the value of an external trigger

Figure 8: a description of notable packet fields, as seen in figure 6

Analysis of Init_TX_ADV_Pkt Function:
We were able to locate this function by searching for references of the TX command structure `rfc_CMD_PROP_TX_ADV_s` identified by the teams of previous semesters. It appears to be integral in populating notable fields such as .pPkt, and.syncword as well as detailing which CS (Carrier Sense) Command to send respective of an inputter function parameter, specifically for the `rfc_CMD_PROP_TX_ADV_s` structure. Motivating the idea that this structure is important to the TX RF transmission scheme.

```
59    Raw_Pkt = (uint)*(ushort *)(Poten_PktPtr + 8);
60    Pkt_Ptr = *(int *)(Poten_PktPtr + 4);
61    FUN_000141a0();
62    FUN_0000f21c_big_to_little_endian(Pkt_Ptr,Raw_Pkt);
63    FUN_000121d0(100,(char)&stack0xfffffff8 + -0x24);
64    if (DAT_20001aac == '\0') {
65      iVar1 = FUN_0000c940(DAT_20001bdb);
66    }
67    else {
68      iVar1 = FUN_0000c940(DAT_20002497);
69    }
70    if (iVar1 == 1) {
71      local_temp = Raw_Pkt + 2 & 0xffff;
72      rfc_CMD_PROP_TX_ADV_s_20002330.pktLen = (uint16_t)local_temp;
73      FUN_000121d0(0xe9,(char)&stack0xfffffff9 + -0x24);
74      if (local_2b == '\x01') {
75        Shifted_Pkt = (byte)((local_temp << 0x15) >> 0x1d) | 0x18;
76      }
77      else {
78        local_temp = Raw_Pkt + 4 & 0xffff;
79        Shifted_Pkt = (byte)((local_temp << 0x15) >> 0x1d) | 8;
80      }
81      *(byte *)(Pkt_Ptr + -1) = Shifted_Pkt;
82      rfc_CMD_PROP_TX_ADV_s_20002330.pPkt = (uint8_t *)(Pkt_Ptr + -2);
83      *rfc_CMD_PROP_TX_ADV_s_20002330.pPkt = (uint8_t)local_temp;
84      rfc_CMD_PROP_TX_ADV_s_20002330.status = 0;
```

Figure 9: Init_TX_ADV Function lines 59-84

Focusing on the population of the memory location of the packet data (declaration on line 82) the data structure from which the local variable is accessing is pulled in from a global *Poten_PktPtr* value (Line 59: `Raw_Pkt = (uint)*(ushort *)(Poten_PktPtr + 8)`). Located at address 200017bc, *Poten_PktPtr's* address is incremented by 8 and stored in a local Raw_Pkt variable.
Utilizing a local variable on lines 71-72 the .pktLen data is pulled out and stored in memory under `rfc_CMD_PROP_TX_ADV_s.pktLen`
Based on value of conditional on line 74, the address representing the packet data is either stripped of the pktLen value (line 78) or left with it, after bits are added flags are "OR'd" in (lines 75 & 79) the value is then stored in memory under rfc_CMD_PROP_TX_ADV_s.pPkt.

```
112      if ((COND_RULE == 1) || (COND_RULE == 4)) {
113        PTR_TX_ADV = (rfc_CMD_PROP_TX_ADV_s *)&rfc_CMD_PROP_CS_s_200022f8;
114      }
115      else if (COND_RULE == 0) {
116        PTR_TX_ADV = (rfc_CMD_PROP_TX_ADV_s *)&rfc_CMD_PROP_CS_s_20002314;
117      }
118      else if (COND_RULE == 5) {
119        Rcv_Adv_Pkt = FUN_0000bc84 + 1;
120        PTR_TX_ADV = (rfc_CMD_PROP_TX_ADV_s *)&rfc_CMD_PROP_CS_s_200022f8;
121      }
122      else if (COND_RULE == 3) {
123        Pkt_Ptr = FUN_0001318c(DAT_20001beb);
124        PTR_TX_ADV = (rfc_CMD_PROP_TX_ADV_s *)&rfc_CMD_PROP_CS_s_200022f8;
125        if (*(char *)(Pkt_Ptr + 0xf) != '\x02') {
126          PTR_TX_ADV = &rfc_CMD_PROP_TX_ADV_s_20002330;
127        }
128      }
129      else {
130        PTR_TX_ADV = &rfc_CMD_PROP_TX_ADV_s_20002330;
131      }
132      Pkt_Ptr = (**(code **)(DAT_200024b8 + 0x50))
133                          (Poten_TX_Callback,PTR_TX_ADV,&local_40,Rcv_Adv_Pkt,2,0);
134      PKT_PTR_TX = (undefined2)Pkt_Ptr;
```

Figure 10: Init_TX_ADV Function lines 112-134

Here the respective CS command is detailed and stored under a local variable based off the function parameter COND_RULE, renamed to represent the *Carrier Sense Conditional Rules* (see below)

Table 23-7. Condition Rules

Number	Name	Description
0	COND_ALWAYS	Always run next command (except in case of ABORT).
1	COND_NEVER	Never run next command (next command pointer can still be used for skip).
2	COND_STOP_ON_FALSE	Run next command if this command returned TRUE, stop if it returned FALSE.
3	COND_STOP_ON_TRUE	Stop if this command returned TRUE, run next command if it returned FALSE.
4	COND_SKIP_ON_FALSE	Run next command if this command returned TRUE, skip a number of commands if it returned FALSE.
5	COND_SKIP_ON_TRUE	Skip a number of commands if this command returned TRUE, run next command if it returned FALSE.

Figure 11: Conditional rules, as specified in the technical manual

With the major fields of rfc_CMD_PROP_TX_ADV_s populated and the specific command structure chosen as to align with an inputted COND_RULE, the return value of Line 132 (FUN_000057d0) is used to populate a global address renamed to PKT_PTR_TX

Analysis of FUN_0000b6a0 Function:
Found in the references of the TX command structure, this function modifies data values involved in the radio operation command structure such as `commandNo`, `startTime`, and `startTrigger`.

```
if ((DAT_20003ae5 == '\0') || ((DAT_20003ae7 & 0x80) != 0)) {
  DAT_2000278c = DAT_2000278c + 1;
  DAT_20003ae5 = '\0';
  FUN_00014608();
}
```

Figure 12: FUN_0000b6a0 lines 15-19

The assumption is that the radio must be set up in a compatible mode (such as proprietary mode) and the synthesizer programmed using `CMD_FS` as referenced via the TI manual.

```
rfc_CMD_PROP_TX_ADV_s_20002330.startTime = 0;
DAT_2000278a = DAT_2000278a + 1;
rfc_CMD_PROP_TX_ADV_s_20002330.startTrigger =
     (_struct_147)((byte)rfc_CMD_PROP_TX_ADV_s_20002330.startTrigger & 0xf0);
```

Figure 13: FUN_0000b6a0 lines 21-24

These lines of code directly modify the data fields of `startTime` and `startTrigger` in the `RFC_CMD_PROP_TX_ADV_s` structure. It sets `startTime`, which is responsible for absolute or relative start time, to 0. In the next line with the data type of `_struct_147`, it contains four data fields: `triggerType`, `bEnaCmd`, `triggerNo`, and `pastTrig`. It modifies `startTrigger` by performing a bitwise AND operation, clearing the first four bits and preserving the next four bits, from least to most significant. The `triggerType`, the first four bits, is retained, while `bEnaCmd`, `triggerNo`, and `pastTrig`, the last four bits, are set to 0

```
iVar2 = (**(code **)(DAT_200024b8 + 0x50))
                  (PTR_DAT_20002690,&rfc_CMD_PROP_TX_ADV_s_20002330,&local_28,&LAB_0000fa20+1,2,
                   0);
```

Figure 14: FUN_0000b6a0 lines 34-36

In these lines, `iVar2` is being set by a function call at address 000057D1h, named `FUN_00057d0`. It passes in six parameters: `Poten_TX_CallBack`, `&RFC_CMD_PROP_TX_ADV_s_20002330` (start of TX structure, `commandNo`), address of a local variable, address of a function, and the literals two and zero.

```
PKT_PTR_TX = (undefined2)iVar2;
if (-1 < iVar2) {
  PTR_rfc_CMD_PROP_RX_ADV_s_200024a4 = (undefined *)&rfc_CMD_PROP_TX_ADV_s_20002330;
  DAT_20002862 = DAT_20002862 + '\x01';
}
```

Figure 15: FUN_0000b6a0 lines 38-41

The `if` statement is checking if `iVar2` is populated with data, which the team suspects is a complete TX data entry combined together after the `FUN_00057d0` call. The next line sets a pointer to a `rfc_CMD_PROP_RX_ADV_s_200024a4`, which leads to the RX structure, to a pointer of the address of the `rfc_CMD_PROP_TX_ADV_S_20002378` structure.
Given the function's involvement in both the TX and RX structure, it may be a good candidate to keep investigating.

Analysis of Shared FUN_000057d0 Function:
6 Parameter function called at the end of FUN_00005d00 (Init_TX_ADV_Pkt) and FUN_0000b6a0. Parameters include a global variable pointing to another callback function, Carrier Sense Command previously determined and held in PTR_TX_ADV variable, Address of empty local variable (free memory space), Packets / TX commands to be sent held in Rcv_Adv_Pkt, and some hardcoded values 2 and 0.

```
6 int FUN_000057d0(ratmr_t *param_1,ratmr_t *POTEN_TX_PKT,ratmr_t **param_3,ratmr_t *param_4,
7                 uint param_5,uint param_6)
```

Figure 16: Fun_000057d0 Header

```
141   Ret_Array[2] = POTEN_TX_PKT;
142   RF_Object_20002fd0.state.pCbSync._0_2_ = *(short *)(Ret_Array + 0xb);
143   *(undefined *)((int)Ret_Array + 0x2e) = *(undefined *)(local_30 + 1);
144   Ret_Array[1] = local_2c;
145   Ret_Array[3] = param_1;
146   Ret_Array[4] = (ratmr_t *)(param_5 & 0x9fffeffd);
147   Ret_Array[5] = (ratmr_t *)(param_6 & 0b00100000);
148   *(undefined *)((int)Ret_Array + 0b00101111) = 0;
149   Ret_Array[6] = (ratmr_t *)0x0;
150   Ret_Array[7] = (ratmr_t *)0x0;
```

Figure 17: Fun_000057d0 Lines 141-150

Given the Shared FUN_000057d0 Function's involvement with the TX structure and its operations, it takes the inputted parameters and combines them into a singular indexable data type along with specifying various RF Rat Module channel configurations.

Analysis of RX Protocol:
The sensors in the device has mini-managers called micro-controllers that create messages. In the re-

ceiving part (RX), various elements, like managing addresses and handling the reception queue, make sure the communication between the device and the system is smooth and trustworthy. The data queue acts like a conveyor belt, moving packages (messages) between the radio frequency core (RF CC1310) and the main brain of your device, the CPU.

As the message travels through the RX chain, which is like a series of stops, certain parts of the message are removed. This "stripping" process isn't limited to just between the CPU and RF; it happens at different points within the device. It's akin to opening a package, extracting what's necessary, and passing along only the vital information. For the semester, our goals is to understand how we found data structures and be able to point to the first entry of queue, code, etc, but also extend this knowledge. By looking at different functions and understanding more and more about the device and how it works, we can we can have a better understanding for our future goal, which is using Scapy to spoof the system.

Variable	Purpose and meaning
pQueue	Pointer to the data queue responsible for transferring data from the RF core to the main CPU. NULL indicates data not stored.
pktConf	Packet confirmation, finalizing operation, and CRC check.
rxConf	Determines whether data is entered into the queue.
bincludeHd	Includes the received header or length byte in the stored packet; otherwise, discards it.
bAppendRssi	Appends an RSSI byte to the packet in the RX queue. Subsequent steps may involve adding a timestamp.
hdrConf	Header configuration, specifying the number of bits, position of the length field, and the number of bits in the length.
addrConf	Address configuration after the header, specifying address size, sync word identifier, number of addresses, and a signed value for length field incrementation.

Figure 18: Important variables and their purpose/meanings:

Analysis of the matchingIndexFinder Function:

```
undefined4 mathingIndexFinder(uint 8or1or?)

{
  uint index;
  undefined4 returnCode;

  index = AddressCount((uint)*(ushort *)8or1or?);
                    /* Return -1 if the index is 0xff */
  if (index == 0xff) {
    returnCode = 0xffffffff;
  }
  else {
    if ((code *)(&PTR_FUN_0001f928+1_200023f0)[index * 3] != (code *)0x0) {
      returnCode = (*(code *)(&PTR_FUN_0001f928+1_200023f0)[index * 3])(8or1or?);
      return returnCode;
    }
                    /* Return -2 if the function dereferenced from the
                       index variable has a NULL value */
    returnCode = 0xfffffffe;
  }
  return returnCode;
}
```

Figure 19: Image of the matchingIndexFinder Function

The matchingIndexFinder function is used for name hash matching. First, it basically converts the passed in string parameter into a unique sequence of numbers, which is done by the hashing algorithm implemented in the AddressCount Function. If the string is blank or empty, then by default there is no match, so the function returns -1 which is equivalent to the signed 2's complement hex number 0xff. Otherwise, the hash is used to index into pointers to functions starting with the mathingIndexFinder+1_200023f0 function. This way, the dereferenced function would return a match of the hash without ever seeing the actual contents. The result of the dereferenced function would be stored into an int variable called "uVar2" and the function would end up returning this variable containing the appropriate hash match. So, it returns -2 if the function dereferenced from the index variable has a NULL value.

Analysis of the AddressCount Function:

```
uint AddressCount(uint target_address)

{
  undefined4 interrupt_info?;
  uint count;
  undefined **loop_pointer;

  interrupt_info? = disable_interrupts();
  loop_pointer = &PTR_LOOP_200023dc;
  count = 0;
  do {
    loop_pointer = loop_pointer + 3;
                    /* Check if ppuVar2 is NULL */
    if (*(ushort *)loop_pointer == 0) break;
                    /* Check for a match between ppuVar2 and 8or1or?
                       If there's a match, truncate the count variable
                       and jump to the LAB procedure  */
    if (*(ushort *)loop_pointer == target_address) {
      count = count & 0xff;
      goto LAB_0001f7b8;
    }
                    /* Increment count by one if none of the conditions are true */
    count = count + 1;
  } while (count < 6);
                    /* If nothing special happens in the while loop,
                       then set count to 255(0xff). */
  count = 0xff;
LAB_0001f7b8:
  enable_interrupts(interrupt_info?);
  return count;
}
```

Figure 20: Image of the AddressCount Function

The function AddressCount is designed to systematically iterate through a set of pointers, inspecting each to discern if any contains data that matches a specified target address parameter. Executed within a loop that iterates six times, the function evaluates two primary conditions: firstly, if the variable at the current address aligns with the target address, it promptly returns the corresponding index; secondly, in the absence of a match after cycling through the set of pointers, the function conclusively returns 255(0xff). During the iterative process, the loop dynamically calculates the address of the subsequent pointer, enforces specific checks such as ensuring the value at the address is not zero and confirming matches with the target address. Importantly, the incrementation of the index within the loop plays a pivotal role in navigating through the set of pointers, effectively facilitating the search for a distinctive data structure. In the event of a successful match, the function provides valuable information about the index where the match occurred; conversely, the return value of 0xff signals the absence of a match, delivering a comprehensive mechanism for systematically querying the specified set of pointers for the target address.

Analysis of the find_address Function:

```
6  undefined ** FUN_0001f54c_find_address(undefined *target_address)
7
8  {
9    undefined4 interrupt_info?;
10   undefined *puVar1;
11   undefined **pointer;
12   int iVar2;
13   undefined **address;
14   undefined *value;
15
16   interrupt_info? = disable_interrupts();
17   iVar2 = 2;
18   pointer = &PTR_DAT_200023c4;
19   do {
20     address = pointer + 3;
21     value = *address;
22     if (value == (undefined *)0x0) {
23       *address = target_address;
24       pointer[5] = (undefined *)0x0;
25       puVar1 = (undefined *)pass_control(0,0,0);
26       pointer[4] = puVar1;
27       if (puVar1 != (undefined *)0x0) {
28 LAB_0001f58a:
29         enable_interrupts(interrupt_info?);
30         return address;
31       }
32       request_failure?();
33     }
34     if ((value == target_address) ||
35        (iVar2 = iVar2 + -1, pointer = address, address = (undefined **)0x0, iVar2 == 0))
36     goto LAB_0001f58a;
37   } while( true );
38 }
```

Figure 21: Image of find_address function

This function's importance stems from the variable that is shared by both this function and the AddressCount function (PTR_DAT_200023c4). The find_address function is similar is to the AddressCount function, except this time, it checks if there is a value that contains the same address as the passed in parameter. If there is a matching address or if the address stores nothing, then return the address. Otherwise, a null address is returned. In lines 20-33, the address to the next pointer is calculated and the data is stored in the "value" variable. So if the value is null, the target_address is stored in the dereferenced address pointer. Afterwards, the fifth element of the pointer is set to null then the pass_control function is called. What is known so far is that the function passes control to another destination function, and it returns the entry point address of the current destination function. The return value (the current destination function address) from pass_control gets stored in the "puVar1" variable. If "puVar1" isn't null, then interrupts are enabled and the address is returned. Otherwise, the request_failure? function would check if there any issue with the pass_control function in its attempt to fetch the entry point address. In lines 34-36. It is shown that there is a conditional statement that checks if the "value" variable is equal to the target_address. If so, we jump to the LAB_0001f58a label to enable interrupts and return the address. Otherwise, the whole procedure restarts until the "iVar2" variable is equal to zero (the address eventually gets returned).

Analysis of functions 000048e4_packet_switch_Green and 00014bd8_GPIO_read_in_RED Given the function's involvement with the RX structure and its operations, it is a critical compo- nent for further analysis.

Analysis of 00002520_packet_processing Function:
The 00002520_packet_processing function is used to process and analyze the data packet received by the camera. As mentioned earlier in this paper, the data transmitted by the camera is stored in a queue, and this function is called on the current data packet in the queue. The information in the data packet is arranged per the IEEE 802.15.4g format, of which there are several modes available.

000: No whitening
001: CC1101 and CC2500 compatible whitening
010: PN9 whitening without byte reversal
011: Reserved
100: No whitener, 32-bit IEEE 802.15.4g compatible CRC (only CC13x0)
101: IEEE 802.15.4g compatible whitener and 32-bit CRC (only CC13x0)
110: No whitener, dynamically IEEE 802.15.4g compatible 16-bit or 32-bit CRC (only CC13x0)
111: Dynamically IEEE 802.15.4g compatible whitener and 16-bit or 32-bit CRC (only CC13x0)

Figure 22: Modes available in IEEE 802.15.4g format

The Wyze Camera uses the last mode (111), which is the most flexible. As shown in the technical manual, the data packet will have a CRC of either 16 or 32 bits. Whitening, the process of making the output evenly distributed, is also available. Whitening causes the radio to output more evenly across its bandwidth, which allows the radio to run at a higher power without breaking FCC guidelines [6]. The overall structure of the function is as follows:

1. Error checking: the function ensures that the inputs are valid, and that it has not timed out or stopped. If there are any error conditions

found, it uses labels to go to the last part of the function, which handles various errors.

2. Setup: the function uses information from the packet header to set up variables and state variables that will be used later during packet processing.

3. Processing: the function uses the previously defined variables to process the payload of the data packet. This function will be explored further later in this paper.

4. Error handling: as mentioned earlier, the final part of the function includes code to handle various errors that occur either in the beginning of the function during error checking or later throughout the function in cases of various exceptions that may occur.

```
error_thunk_check:
            error_thunk?();
LAB_00002830_not_0:
            uVar3 = thunk_FUN_00013d14((int)local_44_dest[0]);
            *(undefined *)(uVar6_packet_header + 0x1d) = uVar3;
            *(undefined *)(uVar6_packet_header + 0x1e) = 0;
            *(char *)(uVar6_packet_header + 0x1f) = local_44_dest[0];
            (*pcVar9)(puVar4);
            FUN_00010018(*(ushort *)(PTR_20003a40 + 10) & 0x10);
LAB_00002870:
            PTR_20002862_Copy = PTR_20002862;
            FUN_00014758();
          }
LAB_00002c50:
          ppuVar2 = local_3c_FUN_001f7c81_ptr;
          (*(code *)local_3c_FUN_001f7c81_ptr[1])();
          piVar8 = *(int **)(PTR_20003a40 + 0x20);
                      /* Setting status of data entry to 0 */
          *(undefined *)(*piVar8 + 4) = 0;
          *(int *)(PTR_20003a40 + 0x20) = *piVar8;
          (*(code *)ppuVar2[2])();
        }
      }
      else {
                      /* 0x3804 is the rf_PROP_RX_ADV_s code
                         and also the code for PROP_ERROR_NO_FS
                         I think this is error checking */
        if (rfc_CMD_PROP_RX_ADV_s_20002378.status == 0x3804) {
          FUN_000141e0();
        }
      }
    }
```

Figure 23: An example of error handling code

```
                if (DAT_20003ae7 == 0x86) {
                  pcVar9 = (code *)PTR_HwIP_something_1+1_200023b0;
                  uVar6_packet_header = DAT_200017bc;
                  if (DAT_200017bc == 0) goto error_thunk_check;
                  goto LAB_00002830_not_0;
                }
```

Figure 24: The error handlers being called within the processing section of the 2520_packet_processing function

Each of these portions of the 00002520_packet_processing function will be explored more in depth in this paper.

1. Error Checking
As mentioned earlier, the first portion of the function checks various error codes to see if any of them are true. In this case, it will call an error handling code. Some notable checks include looking at the status of the radio. The technical manual provides a list of status codes that may be checked before processing a new packet.

Operation finished normally		
0x3400	PROP_DONE_OK	Operation ended normally
0x3401	PROP_DONE_RXTIMEOUT	Operation stopped after end trigger while waiting for sync
0x3402	PROP_DONE_BREAK	RX stopped due to time-out in the middle of a packet
0x3403	PROP_DONE_ENDED	Operation stopped after end trigger during reception
0x3404	PROP_DONE_STOPPED	Operation stopped after stop command
0x3405	PROP_DONE_ABORT	Operation aborted by abort command
0x3406	PROP_DONE_RXERR	Operation ended after receiving packet with CRC error
0x3407	PROP_DONE_IDLE	Carrier sense operation ended because of idle channel (valid only for CC13x0)
0x3408	PROP_DONE_BUSY	Carrier sense operation ended because of busy channel (valid only for CC13x0)
0x3409	PROP_DONE_IDLETIMEOUT	Carrier sense operation ended because of time-out with csConf.timeoutRes = 1 (valid only for CC13x0)
0x340A	PROP_DONE_BUSYTIMEOUT	Carrier sense operation ended because of time-out with csConf.timeoutRes = 0 (valid only for CC13x0)

Figure 25: Status options as specified in the technical manual

In the code of the 00002520_packet_processing function, there are checks for various status codes in order to ensure that the packets are ready for processing. An example is seen in the following image:

```
if ((param_4 == 0 && ppuVar2 == (undefined **)&DAT_00010000) &&
   (rfc_CMD_PROP_RX_ADV_s_20002378.status == 0x3400)) {

if ((rfc_CMD_PROP_RX_ADV_s_20002378.status == 0x3401) ||
   (rfc_CMD_PROP_RX_ADV_s_20002378.status == 0x3404)) {
```

Figure 26: Examples of the 00002520_packet_processing function checking status codes to ensure correct operation

In these lines of code, the function is checking if the radio status is 0x3400, which corresponds to the OK radio status from the technical manual, the 0x3401 status, which corresponds to the Timeout status in the technical manual, meaning that the end trigger occurred before the syncword was found, and the 0x3404 status, which corresponds to the stopped status.

2. Set Up:
While processing the packet, the IEEE 802.15.4g provides a framework for the packet's information. In the setup portion of the function, the packet header is used to determine information about the packet. The framework specifies that 11 bits of the packet will be the length of the packet.

```
          /* Packet_Second_Byte of data is pointing to 20003bef1 - Whitening
                                 Found for CurrEntry (20003be8 + 9) */
Packet_Second_Byte = PTR_LOOP_20003a54[9];
          /* Packet_First_Byte is pointing to 20003bf0 - which is the PktLength
                                 Found from CurrEntry (20003be8 + 8) */
DAT_20003ae5 = 0;
          /* Get the packet length from the first 11 bits of the packet header */
Pkt_Len = (ushort)(byte)PTR_LOOP_20003a54[8] + (Packet_Second_Byte & 7) * 0x100;
```

Figure 27: This image shows where the packet is stored, as well as retrieving the length of the packet from the first 11 bits of the header.

Bit 12 specifies if the CRC is 2 bytes or 4 bytes.

This information is also used later to determine the length of the packet payload.

```
      /* Checks if bit 12 of the header is 0 to find the length of the CRC of the
         packet. Bit 12 is 0, so the CRC is 32 bits. (Header bits go from 0 to 15.) *
if ((Packet_Second_Byte >> 4 & 1) == 0) {
  Pkt_Data_Len = Pkt_Len - 4;
}
else {
  Pkt_Data_Len = Pkt_Len - 2;
```

Figure 28: An image of code checking the length of the CRC and calculating the packet length by removing either 4 bytes for a 32-bit CRC, or 2 bytes for a 16-bit CRC

Finally, bit 15 determines whether the packet contains a payload or if it is just the header.

```
      /* Checks bit 15 of the packet data to see if the frame contains data or just a
         header, bit 15 is 0 so there is data and CRC. The modified Packet length
         checked if it is less than or equal to what is in 00002a18, which appears to
         be the max packet length */
if (((PTR_LOOP_20003a54[9] & 0x80) == 0) && (Pkt_Data_Len <= DAT_2000287c)) {
```

Figure 29: An image of the code checking bit 15 of the packet header to see if the packet contains a payload, or if it is only the packet header.

Therefore, a representation of a possiblepacket header may look as follows:

0000 1000 0000 0111

Figure 30: An image of a possible packet header. In this image, the orange bits represent the length, the green represents the CRC, the purple represents whitening, and the yellow represents whether or not the packet includes a payload or not.

Also included in the setup is a pointer to the data packet queue, which, by following the pointer, will lead to a data packet stored in memory. This data packet can be used as an example to apply the calculations done in the set up and processing stages of the 00002520_packet_processing function.

```
20003bf0 2d 08 61 88 84  Packet                        Packet
         a0 68 05 75 01
         00 50 90 70 d7...

  20003bf0 2d 08           Pkt_Header                  Header

  20003bf2 61 88 84 a0 68  Pkt_data                    Packet Data
           05 75 01 00 50
           90 70 d7 ec 2d...
  20003c1b e0              uint8_t   E0h               RSSI
  20003c1c 60 b9 e9 a6     uint32_t  A6E9B960h         Timestamp
```

Figure 31: An image of the data packet contained in memory.

As seen in the image, the data packet is broken down into multiple parts, including the packet header, which is used in this portion of the 00002520_packet_processing function, as well as the data contained in the packet, the RSSI, and the timestamp.

The packet header is 0x2d08, and this packet header will be used to provide an example of the computations completed in the set up portion of the function. The packet header in binary is 0b0010110100001000. Thus, the first 11 bits, which represent the length of the packet, are 0b000010110100, or 180 in decimal. Note that there are two important lengths that are used throughout the function, the first being the length of the entire packet, and the second being the length of the packet without the CRC.

Bit 12 is 1, meaning that the packet includes a 32-bit CRC. This means that the total length of the packet is 180 bits, and the packet without the CRC is 148 bits. Bit 15 is 0, meaning that the packet contains a payload and is not just the header.

Also included in this portion of the function are setting up other variables in order to later be processed and ensuring that the packet is in the correct format. The set up also includes checking the length of the packet to see if it is less than the maximum payload, and if so, the modified packet without the CRC is passed into another function that changes the order of the bits from big endian to little endian. The function also uses information from the packet header to set up state variables that will be used later throughout the function for calculations.

```
196          puVar4 = PTR_concat_base_data+1_200023ac;
197                /* Set the value at 20003a4c to 5 at some sort of state variable. Likely used to
198                   remove CRC from packet data later in the function */
199          DAT_20003a4c_state = 5;
215             if ((uVar10_packet_header_data & 2) < 2) {
216               if (((uVar6_packet_header & 7) == 2) && (DAT_20003a41 == '\0')) {
217                   /* Copying 1st byte of current packet to local_44_dest */
218                 FUN_00014a68_memcpy(local_44_dest,Data Entry Queue_20003a60.pCurrEntry + 0xd,1);
219                 DAT_20003a4c_state = 6;
```

Figure 32: An example state variable, DAT_2003a4c_state, is set to either 5 (line 199) or 6 (line 219) depending on information from the packet's header.

After setting up the relevant variables, the function then moves to actually processing the packet payload.

3. Processing

As mentioned earlier, this portion of the function uses the information and variables set up in the previous portion of the function to run computations on the data packet itself. This portion of the function starts at LAB_00002a00_Processing, as this is where the variables are reset to their original values.

```
353 LAB_00002a00_Processing:
354                             DAT_20003a43_4 = 4;
355                           }
356                         }
357                         if ((DAT_20001aac == '\0') || (uVar12 < 2)) {
358                           FUN_00013e38();
359                         }
360                         local_34 = (uint)DAT_20003a43_4;
361                         Transmit_Pkt_Len = (ushort)DAT_20003a45_Pkt_Dat_Len;
362                       }
363                   /* uVar9_prefix = 20h */
364                       packet_first_byte = DAT_20003a50_packet_first_byte;
365                       uVar9_prefix = DAT_20003a4a_prefix;
```

Start of processing

Figure 33: An image of the beginning of LAB_00002a00_Processing, and where the payload processing portion begins.

As seen in the image, on lines 360 and 364, local variables Transmit_pkt_length and packet_first_byte are reset to their corresponding data variables. This is necessary as earlier during the set up portion of the 00002520_packet_processing function, they may have been set to other variables as part of the calculations that were done. Resetting these variables to their corresponding data variables points to their values being reset for this portion of the function.
Each line of this section of the 00002520_packet_processing function was then analyzed individually, keeping note of the updated values. This was done by using the packet data stored in memory, as seen in a previous image, in order to explore how the code is processing the information stored in the data packet.
The following image is a snippet from the processing portion of the 0002520_packet_processing function that contains a few lines of code that will be worked through as an example.

```
packet_first_byte = DAT_20003a50_packet_first_byte;
uVar9_prefix = DAT_20003a4a_prefix;
puVar6 = DAT_20003a50_packet_first_byte + 28;
DAT_20003a50_packet_first_byte[1] = puVar6;
*(undefined2 *)(packet_first_byte + 2) = DAT_20003a48;
```

Figure 34: Image of the beginning of the processing portion of the 00002520_packet_processing function

The first and second line reset the packet_first_byte and uVar9_prefix local variables to their original data values, as mentioned earlier. The third line sets puVar6 to the value of the packet_first_byte data variable plus 28. At this point the packet_first_byte holds 0x2d, which corresponds to 45 in decimal. $45 + 28 = 73$, so puVar6 now holds a value of 73.
On the fourth line, the local variable DAT_20003a50_packet_first_byte variable at index 1 is set to puVar6, which was just calculated to be 73. 73 in binary is 0b1001001. Changing the value at index 1 results in the packet header becoming 0b0111011010001000. This translates to 0x7688.
This process was followed for other lines in the 00002520_packet_processing function. All of the code worked through thus far can be found here.

4. Error Handling
The final component of the function includes error handlers that are called whenever the function runs into an error or exception case during its error checking, set up, or processing phases.

```
if (DAT_20003ae7 == 0x86) {
  pcVar9 = (code *)PTR_HwIP_something_1+1_200023b0;
  uVar6_packet_header = DAT_200017bc;
  if (DAT_200017bc == 0) goto error_thunk_check;
  goto LAB_00002830_not_0;
}
```

```
error_thunk_check_interrupts:
                    /* Enables interrupts and saves data */
          error_thunk?();
```

Figure 35: Example of a call to an error handler (top) and its corresponding handler code (bottom).

Many of the error handlers in this function involve calls to enable interrupts. The interrupts available can be found in the technical manual:

Operation finished normally		
0x3400	PROP_DONE_OK	Operation ended normally
0x3401	PROP_DONE_RXTIMEOUT	Operation stopped after end trigger while waiting for sync
0x3402	PROP_DONE_BREAK	RX stopped due to time-out in the middle of a packet
0x3403	PROP_DONE_ENDED	Operation stopped after end trigger during reception
0x3404	PROP_DONE_STOPPED	Operation stopped after stop command
0x3405	PROP_DONE_ABORT	Operation aborted by abort command
0x3406	PROP_DONE_RXERR	Operation ended after receiving packet with CRC error
0x3407	PROP_DONE_IDLE	Carrier sense operation ended because of idle channel (valid only for CC13x0)
0x3408	PROP_DONE_BUSY	Carrier sense operation ended because of busy channel (valid only for CC13x0)
0x3409	PROP_DONE_IDLETIMEOUT	Carrier sense operation ended because of time-out with csConf.timeoutRes = 1 (valid only for CC13x0)
0x340A	PROP_DONE_BUSYTIMEOUT	Carrier sense operation ended because of time-out with csConf.timeoutRes = 0 (valid only for CC13x0)

Figure 36: A chart of the possible interrupts that may be called.

6 Conclusion

The purpose of this research is to gain a better understanding of how the Wyze Camera transmits and receives information. By reverse engineering the camera's TX, RX, and packet-processing protocols, we can learn more about what information the camera expects to receive and transmit. With this information, we can find potential vulnerabilities by looking to see how the camera reacts to bad or malformed data. Using this information, we can create a baseline for how we expect the camera to react to different data.
Since manually reverse engineering a process can be extremely time consuming, it is also very important to try to use an automatic process to learn more about a device's vulnerabilities. This can be accomplished through a process called fuzzing. In fuzzing, a computer continuously and automatically sends poor data to an external device while monitoring and recording its output to the data [7]. This

allows for a constant monitoring of the device's reactions to the data, which can then be analyzed to look for exceptions, crashes, and other signs of vulnerabilities. Since this process occurs automatically, it is a much more efficient way to find vulnerabilities than manually reverse engineering the device.

The end goal of this research project is to use the results of the manual reverse engineering to create a baseline for the Wyze Camera's output to various data forms. Then, by fuzzing the data and comparing the output of this to the results from manually reverse engineering the data, the research team can learn more about whether or not fuzzing is a viable process for finding vulnerabilities, which will greatly increase the efficiency of finding vulnerabilities in IoT devices.

7 References

[1] Peters, J. (2023, September 8). Your wyze webcam might have let other owners peek into your House. The Verge. https://www.theverge.com/2023/9/8/23865255/wyze-security-camera-feeds-web-view-issue

[2] 8 internet of things threats and risks to be aware of. SecurityScorecard. (2021, August 4). https://securityscorecard.com/blog/internet-of-things-threats-and-risks/

[3] Cybtech. (2022, September 22). Home security: Why you should put IOT devices on a guest Wi-Fi Network. Dual Layer IT Solutions LTD. https://www.duallayerit.com/home-security-why-you-should-put-iot-devices-on-a-guest-wi-fi-network/).

[4] Texas Instruments, "Cc13x0, cc26x0 simplelinkTM wireless mcu technical reference manual," Texas Instruments, Feb 2015.

[5] Security vulnerabilities identified in Wyze Cam IOT device - bitdefender. Bitdefender. (2022, March 29). https://www.bitdefender.com/files/News/CaseStudies/study/413/Bitdefender-PR-Whitepaper-WCam-creat5991-en-EN.pdf

[6] Christiansen, G. (n.d.). DN509 – Data Whitening and random TX mode. https://www.ti.com/lit/an/swra322/swra322.pdf

[7] What is Fuzz Testing and how does it work?. Synopsys. (n.d.). https://www.synopsys.com/glossary/what-is-fuzz-testing.html