Wyze Paper Spring 2024

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

February 2024

1 Abstract

This paper covers the work done on the Wyze IP Camera by the Embedded Systems Cyber Security VIP at Georgia Institute of Technology. The paper will include work up to and including Spring 2024. The VIP Project's goal is to manually reverse engineer the wireless communications protocol portion of the Wyze IP Camera in order to find any vulnerabilities in its code. The main areas of focus in the wireless protocol for this semester's research include the camera's methods of receiving, transmitting, and processing packets. The camera's binary was disassembled in Ghidra to provide psuedo-C code, which is being used to learn more about each of these sections. In the future, the results found from manually reverse engineering the camera will be compared to the results of automatically spoofing the camera, in order to determine if automatic spoofing is a viable candidate for finding vulnerabilities in RF devices.

2 Introduction

The Wyze IP Camera is an IoT (Internet of Things) device which allows users to see a camera feed that they are not physically present for. The Wyze Camera has both a web application (Wyze Web View) and a mobile application (Wyze App). After setting up the camera and connecting to it, users can then view the camera's feed at any point in time through the web app or the mobile application. This process includes first physically setting up the camera in the preferred location and creating an account on their personal devices. The user will then pair the camera and connect it to their account. After the set up is complete, users will be able to view remote footage from the camera on their personal devices. The Wyze camera is advertised as a camera for homes, rental properties, and businesses [1].

The purpose of the team's research is to reverse engineer the firmware from the Wyze IP Camera's Sensor Bridge. The team is specifically trying to gain information regarding how the camera sends, receives, and processes information in order to form and send malformed data to the camera. From this, the team can see how the camera behaves when it encounters malformed data, which could reveal information about the camera's vulnerabilities.

Previously, the Embedded Systems Cyber Security Wyze Team has been able to physically take apart the camera and reverse engineered its function. The team has previously set up a Man in the Middle attack in order to get the firmware and utilized debug ports in order to exfiltrate the camera's memory state. The team has also been able to perform a replay attack on the device and perform memory captures on the camera. In this semester's research, the team is using the memory capture in order to further reverse engineer the camera's process of data manipulation while receiving, sending, and processing data.

3 Device Description

The Wyze Camera is able to utilize data from two sensors, the contact sensor and the motion sensor. See Figure 1 for the Contact Sensor (left) and Motion Sensor (right) hardware. It also includes an SD card slot to save footage.

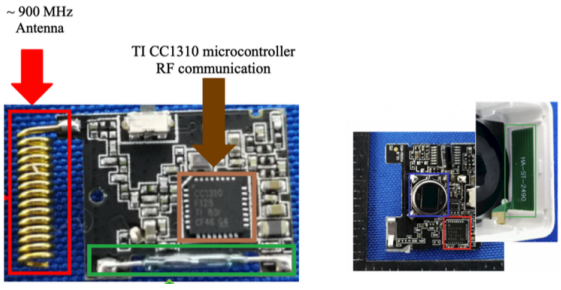

Figure 1: Peripheral Sensor Hardware

These peripheral sensors communicate with the sensor bridge via proprietary RF communication. The sensor bridge is the component that is responsible for connecting peripheral sensors to the device's mi-

crocontroller. The sensor bridge then communicates with the camera through a USB form factor. The camera's processors finally use IEEE 802.15.4 wi-fi in order to send their data to the Wyze user application, which is available both as a mobile app and as an online web service.

Figure 2: Wyze Camera System Communication Overview

FSK (Frequency Shift Keying) is used to communicate between the peripheral sensors and the sensor bridge, where FSK supports data rates of 625 bps to 4 Mbps. It can also perform Minimum Shift Keying (MSK) and On-Off Keying (OOK) Modulation. The TI CC1310 utilizes two processors (ARM Cortex M3 & M0), as well as peripheral controllers.

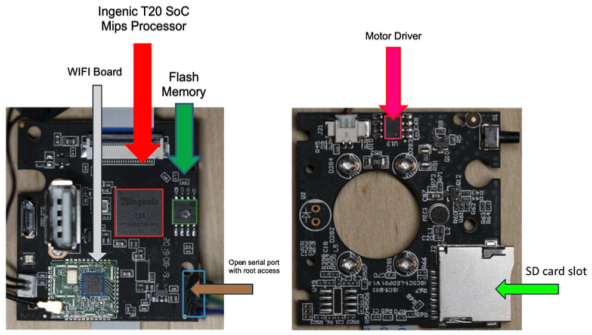

Figure 3: Wyze Camera Hardware

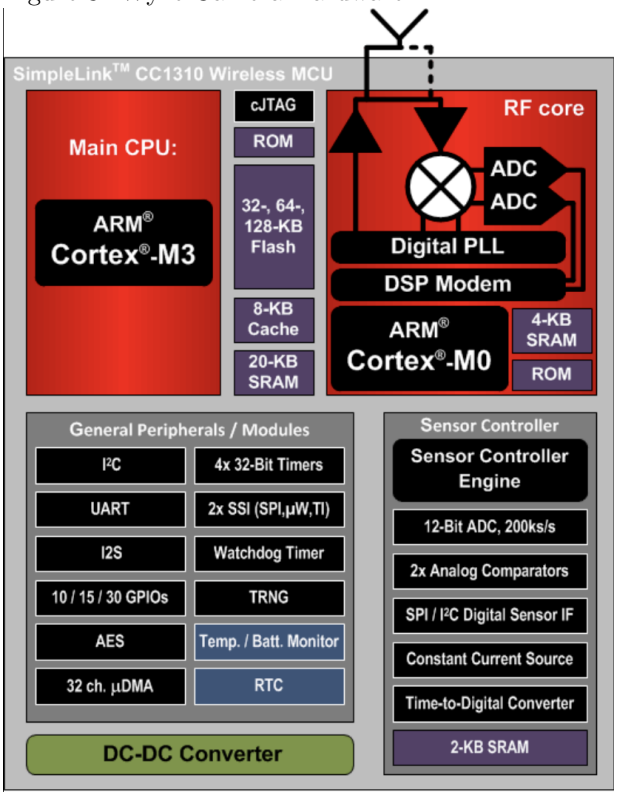

Figure 4: TI CC1310 Chip Overview [2]

The first processor, the ARM Cortex M3, is the main processor. The M3 processor is part of the the system-side and runs the user application, operates on the information from the TX and RX packets. The M3 utilizes a 8Kb Cache and 20Kb SRAM as seen in Figure 4. 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, similar to the M3 the M0 has a 4Kb SRAM as seen in Figure 4.

The Wyze Camera implements a Ingenic T20 processor running Busybox Linux. On boot, the Wyze Camera configurs the basic system environment before other services start through `/etc/init.d/rcS`. Later calling/system/bin/iCamera and /system/bin/dongle_app where `iCamera` is the main utility and `dongle_app` manages the communication with the sensor bridge.

"The (cc1310) RF core receives high-level requests from the system CPU and performs all the necessary transactions to fulfill them. These requests are primarily oriented to the transmission and reception of information through the radio channel, but can also include additional maintenance tasks such as calibration, test, or debug features. [2]"

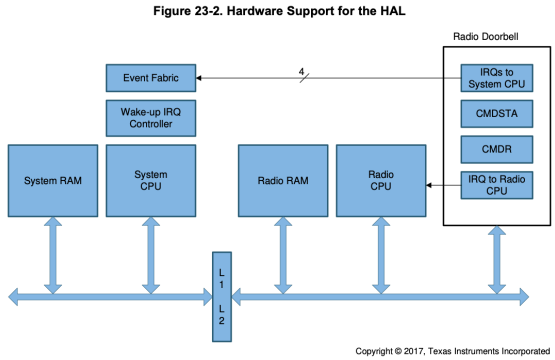

Figure 5: CC1310 Hardware Support for the HAL [2]

A radio doorbell module (*known as a CPE "command and packet engine"*) is utilized as the primary means of communication between the system and radio CPU. Where parameters and payloads are transferred through the system & radio RAM and during operation the radio CPU updates parameters and payloads in its RAM and raises interrupts. Specifically, commands are sent to the radio through a CMDR register and the radio CPU is notified whenever a value is written to this register. After processing the value in this register, a RFCMDACK interrupt is raised and mapped to RFACKIFG register.

There are 3 types of commands able to be issued through the CMDR register:
1. Radio Operation Commands
2. Immediate Commands
3. Direct Commands

For Radio Operation and Immediate Commands the CMDR register contains a pointer to the command structure w/ the 2 LSBs set to 0.

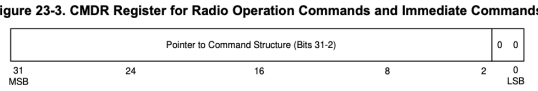

Figure 6: Radio Operation and Immediate Command Structure [2]

For Direct Commands the 2 LSBs are set to 01, a command ID set in bits 16-31, and optional parameters in bits 2-16.

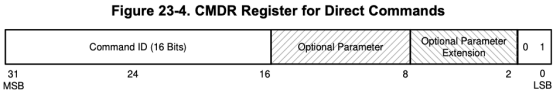

Figure 7: Direct Command Structure [2]

In the sense of context for the later TX analysis of "Erro_Check_StartTrigStartTime" the RF core has a dedicated timer module (Radio Timer = RAT) that is used in execution of Radio Operation commands w/ delayed starts or any that runs the reciever or transmitter. The RAT allows for Compare and Capture events that respectfully raise interrupts on when the Radio counter reaches a certain compareTime and to record the time of transitions of input pins.

4 Existing Vulnerabilities

Research teams have discovered weaknesses within the Wyze Camera. These include ways to get around the authentication and buffer overflows in the Input/Output control of the Wyze Camera. To log into the cellphone application, you need to have a username and password in order to authenticate. To link the app and camera together, the camera then transmits a code to the user's phone. A flaw in this prevents the authentication code from actually being kept in the camera's memory; instead the NULL value that is originally there is retained. The camera will then be connected without actually receiving the authentication code as long as the NULL value is entered as the code [3]. Also, though a password is needed to authenticate the camera, it can be cracked, especially when the password is weak, as many users may have.[4]

A buffer overflow during the authentication process has also been found inside the IOCtl (Input/Output Control) [3]. When there is an input, the size of the payload is included. A big payload exceeds the buffer and writes outside of the space allotted because the camera does not compare this value to the size of its destination buffer.

As of February 16 of 2024, Wyze also reported that there was a technical glitch with AWS. 13,000 user clients saw other user's homes[4] because of a third-party app that would cache the client library of the camera, and it had mixed up the Device ID and the User ID [3].

5 Reverse Engineering Results

RX Analysis

The sensors in the device has mini-managers called micro-controllers that create messages. In the receiving 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 8: rfc_CMD_PROP_RX_ADV_s Command Fields **Analysis of matchingIndexFinder**

```
int mathingIndexFinder(uint 8)

{
  uint index;
  int returnCode;
  int iVar1;

  index = AddressCount((uint)*(ushort *)8);
                    /* returns 0xff, which is index

                       If the function went through the loop without finding any target address
                       matches, index will be 0xff and match the first if check */
                    /* Return -1 if the index is 0xff */
  if (index == 0xff) {
    returnCode = -1;
  }
  else {
    if ((code *)(&PTR_FUN_0001f928+1_200023f0)[index * 3] != (code *)0x0) {
      iVar1 = (*(code *)(&PTR_FUN_0001f928+1_200023f0)[index * 3])(8);
      return iVar1;
    }
                    /* Return -2 if the function dereferenced from the
                       index variable has a NULL value */
    returnCode = -2;
  }
                    /* returns 0xffffff, which is ?-1 */
  return returnCode;
}
```

Figure 9: matchingIndexFinder Function

The function mathingIndexFinder is used for determining an index based on the input parameter and performs actions based on this index. The index is found by delving into the function AddressCount which operates on a presumed address converted from the input. If the obtained index is 0xff, the function returns -1, indicating no match was found. If the index is valid, the function checks a function pointer array (likely for handler functions), dereferencing the function at the calculated array position using the index. If the dereferenced function pointer is non-null, it calls this function with the input parameter and returns its result. In our case, since we know from 000014f0_seems_important_packetthat the input is an int 8 (local8), all input variables will be 8. If the function pointer is null, it returns -2, indicating an error due to a null function pointer. The function deals with mapping input values to specific function handlers, handling errors, and undefined handler functions.

Analysis of AddressCount

```
uint AddressCount(uint 8)

{
  undefined4 interrupt_info?;
  uint count;
  byte 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 == 8) {
      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 10: addressCount Function

The function AddressCount is a function that iterates through a list of addresses to find a specific address depending on the input, which in our scenario is 8. AddressCount function returns two things:

$$\text{count} = \text{count} \ \& \ 0\text{xff}$$
$$\text{count} = 0\text{xff};$$

This totally depends on what the loop pointer is. After delving more into the function we can actually find the array values of PTR_LOOP_200023dc, and what value we get after iterating through it a few times:

```
200023dc 44 02 00 20     addr        PTR_LOOP_20000244
200023e0 e8 2d 00 20     addr        DAT_20002de8

                    DAT_200023e4
200023e4 00 00 00 00     undefined4  00000000h

                    DAT_200023e8


200023e8 08 00           undefined2  0008h
```

Figure 11: The address values around &PTR_LOOP_200023dc.

We know that a single iteration inside the for loop means a 3 byte addition for our loop_pointer. This means after a single count iteration, loop_pointer will be pointing to 200023e8, which is 0008h. We now know that _loop_pointer = 8. Due to the if statement inside, we take exit after count = count & 0xff and since count is 1, we just print out 0xff. This way we know count will stay one (1 & 11111111 = 1) and the function will return that.

Analysis of 000048e4_packet_switch Green function

The function 000048e4_packet_ switch_Green is a packet processing routine with lots of switch statements that determine the specific operation to be executed.We don't know if this is a state machine or just a bunch of nested ifs. Until we know for sure, we are going to call it a routine with logic to determine. It handles a variety of packet-related cases, including packet creation, manipulation, and forwarding. Depending on these 16 cases, the function performs tasks such as copying data, calling other functions, and interacting with specific memory locations. The function is part of `000014f0_seems_important_packet`, located at the bottom of the tree where it's affiliated with another method we've worked on the previous semester `PTR_mathingIndexFinder` (in 200023a8).

```
    puVar2 = &local_58;
    break;
  case 4:
    if (*(int *)(PTR_DAT_2000284c + 0xc) == 0) {
      return;
    }
    0000f934_GREEN_edit_value(&local_58,0,0xe);
    local_58 = CONCAT31(local_58._1_3_,*(undefined *)(0? + 1));
    FUN_00014978((int)&local_58 + 2,0? + 2);
    local_4c = *(undefined2 *)(0? + 0xc);
    pcVar5 = *(code **)(puVar3 + 0xc);
    puVar2 = &local_58;
    break;
  case 5:
    if (*(int *)(PTR_DAT_2000284c + 0x10) == 0) {
      return;
    }
    FUN_0000d630(0?);
    return;
  case 6:
    if (*(int *)(PTR_DAT_2000284c + 0x14) == 0) {
      return;
    }
    0000f934_GREEN_edit_value(&local_58,0,0x13);
    local_58 = *(uint *)(0? + 2);
    local_54 = *(uint *)(0? + 6);
    FUN_0000e23c_memcpy(&local_50,(void *)(0? + 10),0xb);
    pcVar5 = *(code **)(puVar3 + 0x14);
    puVar2 = &local_58;
    break;
  case 7:
    if (*(int *)(PTR_DAT_2000284c + 0x18) != 0) {
      FUN_000118d4(0?);
```

Figure 12: Cases 4 through 7, a showcase of what the function looks like, as well as the 4 addition each case.

Another interesting thing about the switch cases is that each time a case statement is checked, the next one will be checked with an additional +4 value. Starting from case 1, we simply check if PTR_DAT_2000284c == 0. Then, for the second case, we check PTR_DAT_2000284c + 4 == 0. For third, PTR_DAT_2000284c + 8 == 0, etc. The function increments the value of a byte to check different addresses if they equall 0.

The 000048e4_packet_switch_Green function takes in a parameter of 0, which is initialized at 000014f0_seems_important_packet. On one scenario it could also be 2, but that if statement is not being passed through, also the value being 2 would make no sense. So, we decided on the value 0.

```
puVar3 = PTR_DAT_2000284c;
puVar2 = &local_58;
if (PTR_DAT_2000284c == (undefined *)0x0) {
  return;
}
switch(*(undefined *)0?) {
case 1:
```

Figure 13: 000048e4_packet_switch_Green lines 24-30

Lines 24 − 30 provide an opening for this function. We know that it skips the if statements and starts the 16 switch cases. This is a dereferencing of a dereferencing. The first dereference takes to location of array where value is located where master stack is at (At the top of the file). Second dereference takes us to stack_begin 2 dereferences.

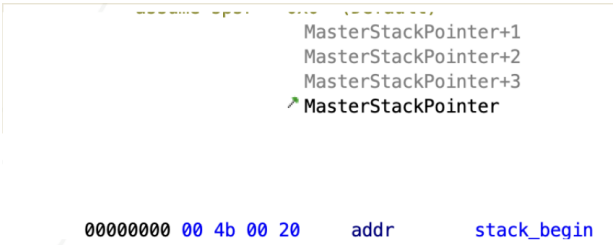

Figure 14: MasterStackPointer.

This is where we get after the first dereferencing, and after the second one, it becomes stack_begin which is BEh (hexadecimal, BE). After this, we start delving into 16 switch cases.

The main switch case we've been working on was case 7, and that is because case 7 checks if PTR_DAT_2000284c does NOT equal to 0, which makes more sense. Every other switch case checks if PTR_DAT_2000284c + x == 0.

```
case 7:
  if (*(int *)(PTR_DAT_2000284c + 0x18) != 0) {
    FUN_000118d4(0?);
  }
  if (*(char *)(0? + 2) == '\0') {
    iVar4 = *(int *)(0? + 0x18);
  }
  else {
    iVar4 = *(int *)(0? + 0x18);
  }
  if (iVar4 == 0) {
    return;
  }
  FUN_00013fe0();
  return;
```

Figure 15: Case 7 of the packet_switch_Green Function

Case 7 first checks if the data + 18 does not equal to 0 (which we believe holds), then jumps into FUN_000118d4(0). This is where we left off this semester and are planning on continuing in the future.

```
void FUN_000118d4(int 0?)

{
  uint address?;
  undefined auStack_24 [17];
  char nullChecker;
  int local_10;

  0000f934_GREEN_edit_value(&address?,0,28);
  address?._0_1_ = *(undefined *)(0? + 1);
  address?._1_1_ = *(undefined *)(0? + 2);
  address?._2_1_ = *(undefined *)(0? + 3);
  address?._3_1_ = *(undefined *)(0? + 4);
  FUN_0000e23c_memcpy(auStack_24,(void *)(0? + 5),0x11);
  nullChecker = *(char *)(0? + 0x16);
                    /* if NULL
                       */
  if (nullChecker != '\0') {
    local_10 = *(int *)(0? + 0x18);
  }
  (**(code **)(PTR_DAT_2000284c + 0x18))(&address?);
  return;
}
```

Figure 16: FUN_000118d4 for showcase

Another thing we observed was that another variable called local_58 (refer to figure 12) frequently appeared in most switch cases. Whenever a switch case contains this variable, it either gets assigned to an accessed piece of data after param_1 was typecasted and dereferenced, or it gets assigned to a concatenation of local_58 and some section of param_1.

```
local_58 = CONCAT22(*(undefined2 *)(param_1 + 2),(undefined2)local_58);
local_58 = CONCAT31(local_58._1_3_,param_1[1]);
```

We also see that local_58 gets concatenated with param_1 in two ways. The first method concatenates the upper 2 bytes of "param_1 + 2" with the upper 2 bytes of local_58 while the second method concatenates the upper 3 bytes of local_58 with the second byte of param_1.

TX Analysis:

Packets from the Wyze camera are composed of a 1bit to 32 byte Preamble, 8 to 32 bit Syncword, 0 to 32 bit header, 0 to 8 byte address, arbitrary size payload, and a 0 to 16/32 bit CRC (see Figure 16). We know that the preamble is a series of alternating 1's and 0's used to gain control of the receiving/transmission link, syncword is a unique sequence used by the packet engine to detect the start of the packet, header containing some packet metadata, and a CRC (cyclic redundancy check) used to verify the message's integrity. The cc1310 allows for multiple modes of radio transmission, however the Wyze developers chose to operate using it's proprietary mode. In order to prepare for packet transmission using cc1310's proprietary mode, the radio must be set up with the CMD_PROP_RADIO_DIV_SETUP command. To transmit a packet the TI CC13x0 utilizes the CMD_PROP_TX_ADV command. [2]

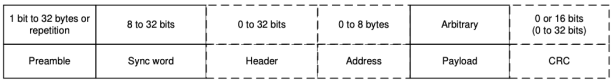

Figure 17: Advanced Packet Format [2]

Building off the previous semester's work, references to the rfc_CMD_PROP_TX_ADV_s command structure was used to identify functions that cooresponded to RF Core operations.

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 18: rfc_CMD_PROP_TX_ADV_s Command Fields:

Certain fields of this command structure were important to identify as they corresponded to important packet processing functionality. Using pPkt we can idenfy the data being sent between modules, preTrigger is helpful in determining the length of preamble bytes and where the syncword starts in packet transmission, pktLen / numHdrBits we can use to indentify the length of payload / header data in a capture of RF traffic, and pktConf & startConf are useful in determining how the specific packet was manufactured and what the redundancy checks encompass.

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 19: a description of packet fields [2], as seen in Figure 16

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. Where this specific function pulls a global pointer to a potential TX packet, checks for any interrupts blocking the creation of the packet, and after some error checking on the pointer and it's defenced value populates the rfc_CMD_PROP_TX_ADV_s.pPkt, rfc_CMD_PROP_TX_ADV_s.status, and rfc_CMD_PROP_TX_ADV_s.startTrig fields. Utilizing a local variable .pktLen data is pulled out and stored in memory under `rfc_CMD_PROP_TX_ADV_s.pktLen` In addition, the respective CS command is detailed and stored under a local variable 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 18: Conditional rules [2]

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 (FUN_000057d0) is used to populate a global address renamed to PKT_PTR_TX

Analysis of Shared FUN_000057d0 Function:
6 Parameter function called at the end of FUN_00005d00 (Init_TX_ADV_Pkt) and FUN_0000b6a0. Passed into this function is a global variable pointing to a callback function, Carrier Sense Command previously determined, Address of empty local variable (free memory space), Packets / TX commands to be sent, 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 20: Fun_000057d0 Method Signature

The primary purpose of this Fun_000057d0 is to compile the fields of a CMD_PROP_TX_ADV command and return a pointer to the compiled struct (known locally to this function as Ret_Array)

In the later conditionals of Fun_000057d0, a function "Error_Check_StartTrigStartTime" is used to determine a RF_RatModule channel mode, specifying the RF_RatMode to be either Compare (RF_RatMode = 1, line 81) or Capture (RF_RatMode = 2, line 97)

```
73    iter_startTime = (ratmr_t **)Find_RF_cmdFs_startTime(RF_cmd?);
74                  /* RF_Object_20002fd0.state.mode_state.cmdFs._12_4_ denotes
75                     RF_Object_20002fd0.state.mode_state.cmdFs.startTrigger @ location 20003008 */
76    if (((iter_startTime == (ratmr_t **)0x0) &&
77        (RF_Object_20002fd0.state.mode_state.cmdFs.startTime != 0)) &&
78       (bVar1 = Error_Check_StartTrigStartTime
79                          ((int)Ret_Array,(uint8_t)RF_Object_20002fd0.state.mode_state.cmdFs._12_4_,
80                           RF_Object_20002fd0.state.mode_state.cmdFs.startTime), bVar1 != 0)) {
81      RF_RatModule_s_2000310c.channel[0].mode._1_1_ = 1;
82      *Ret_Array = (ratmr_t *)RF_Object_20002fd0.state.mode_state.cmdFs.startTime;
83                  /* Stores the value of ppiVar3 into 20003004 */
84      RF_Object_20002fd0.state.mode_state.cmdFs.startTime = (ratmr_t)Ret_Array;
85    }
86    if (RF_RatModule_s_2000310c.channel[0].mode._1_1_ == 0) {
87      bVar1 = 0;
88      temp_strtTime = (ratmr_t **)RF_Object_20002fd0.state.mode_state.cmdFs.startTime;
89      if (iter_startTime != (ratmr_t **)0x0) goto LAB_00005896;
90                  /* Iterate until strtTime_ptr and iter_startTime trigger FUN_000d584 to return
91                     != 0, then save the iter_startTime to strtTime_ptr and set Timer channel to
92                     mode 2 and record the address of the channel mode */
93      while (iter_startTime = temp_strtTime, iter_startTime != (ratmr_t **)0x0) {
94  LAB_00005896:
95        bVar2 = Error_Check_StartTrigStartTime
96                          ((int)Ret_Array,(uint8_t)iter_startTime,(ratmr_t)*iter_startTime);
97        if (bVar2 != 0) {
98          RF_RatModule_s_2000310c.channel[0].mode._1_1_ = 2;
99          *Ret_Array = *iter_startTime;
100         bVar1 = RF_RatModule_s_2000310c.channel[0].mode._1_1_;
101         *iter_startTime = (ratmr_t *)Ret_Array;
102         break;
103       }
104       temp_strtTime = (ratmr_t **)*iter_startTime;
105     }
```

Figure 21: 1st and 2nd use of Error_Check_StartTrigStartTime

Here we check the validity of this a located startTime (through function call on line 73) and RF_Object startTime with conditional (line 76) calling Error_Check_StartTrigStartTime, if this passes we set the RF_RatModule channel mode to 1 (RatModeCompare) and set the respective dereferenced RetArray value to this startTime. Then, if the RF_RatModule is 0 (RatModeUndefined) we set a local temp_strtTime to the value of the RF_Object start time and check the validity of this value (conditional on line 89). If this passes we enter a while loop checking startTime values until one returns with no error

(Error_Check_StartTrigStartTime check on line 95). On the first Error_Check_StartTrigStartTime no error (we have a valid startTime and startTrigger), we set the RF_RatModule channel mode to 2 (RatModeCapture) and store the startTime in the Ret_Array.

```
106     if (bVar1 == 0) {
107       iVar3 = FUN_0000b854(local_28,*(undefined *)(local_30 + 1));
108       if (iVar3 == 1) {
109         RF_RatModule_s_2000310c.channel[0].mode._1_1_ = 4;
110       }
111       else if ((RF_RatModule_s_2000310c.channel[0].mode._1_1_ == 0) && (iVar3 != 1)) {
112         iter_startTime = (ratmr_t **)RF_Object_20002fd0.state.mode_state.cmdFs.startTime;
113         if (RF_Object_20002fd0.state.mode_state.cmdFs.startTime != 0) {
114           for (; *iter_startTime != (ratmr_t *)0x0; iter_startTime = (ratmr_t **)*iter_startTime)
115           {
116           }
117         }
118         if (((*(byte *)(POTEN_TX_PKT + 3) & 0xf) == 2) &&
119           ((int)(uint)*(byte *)(POTEN_TX_PKT + 3) >> 7 == 0)) {
120           if (iter_startTime == (ratmr_t **)0x0) {
121             if (RF_Object_20002fd0.state.mode_state.cmdFs._12_4_ == 0) goto LAB_00005924;
122             bVar1 = Error_Check_StartTrigStartTime
123                               ((int)Ret_Array,
124                                (uint8_t)RF_Object_20002fd0.state.mode_state.cmdFs._12_4_,0);
125           }
126           else {
127             bVar1 = Error_Check_StartTrigStartTime((int)Ret_Array,(uint8_t)iter_startTime,0);
128           }
129           if (bVar1 == 0) goto DetermineChannel;
130         }
131       }
132     }
```

Figure 22: 3rd use of Error_Check_StartTrigStartTime

If a proper startTime is not found in the previous while loop, the RF_RatModule channel mode is set to 4 (not a documented RF_RatMode) based off return value of FUN_0000b854, else if the the current channel is 0 and $iVar3! = 1$ we enter an empty for loop (possibly just iterating to a populated startTime field). Continuing, Error_Check_StartTrigStartTime is called twice to validate the current value of startTime, and if there is an error we retry finding the startTime through the use of a DetermineChannel label (points back to line 69).

Later in Fun_000057d0 there are handlers for each of the specified RAT channel modes (Undefined, Compare, Capture) as well as a handler for a developer added mode aptly named "Custom". As noted in the Technical manual *"In Compare Mode, the timer generates an interrupt when the counter reaches the value given by compareTime. The interrupt is mapped to RFHWIFG"* and for Capture Mode *"When the transition occurs, the current value of the RAT is stored in the RATCHnVAL register corresponding to the selected channel and the timer generates an interrupt"*. However the handler for Undefined mode checks for some undelt data located in the Ret_Array struct and sets bits in the command and Ret_Array struct cooresponding to this scenario.

```
162   if (RF_RatModule_s_2000310c.channel[0].mode._1_1_ == 4) {
163     00015176_interacts_with_radio_GREEN
164               (RF_RatModule_s_2000310c.channel[0].pClient,
165                (int)(short)RF_RatModule_s_2000310c.channel[0].pCb,
166                (uint)RF_RatModule_s_2000310c.channel[0].pCb._2_1_);
167   }
```

Figure 23: Custom Rat Mode Handler

In the case that the Rat channel mode == 4 (not a documented value) we call an encapsulation of function aptly named "customRatFun" w/ a passed in client struct, callback function, and specified flags cooresponding to the client.

```
2 undefined4 customRatFun(RF_Handle pClient,int pCb,uint pCbFlags)
3
4 {
5   undefined4 uVar1;
6   int iVar2;
7   ratmr_t rVar3;
8   uint32_t pOp;
9   undefined4 uVar4;
10  int iVar5;
```

Figure 24: customRatFun Method Signature

Here we validate the callback function, and calculate an offset to the callback function. After validating this offset we then specify a length field and utilize the doorbell module to send some data *(further analysis required)*

```
15  if (pCb == -1) {
16                    /* if callback is undetermined
17                       HashMatch pClient and startTrigger */
18    iVar2 = HashMatch(pClient,RF_Object_20002fd0.state.mode_state.cmdFs._12_4_);
19                    /* if objects are not equal, set rVar3 to startTrigger */
20    rVar3 = RF_Object_20002fd0.state.mode_state.cmdFs.startTime;
21                    /* TDCCLKCTL clock control?  */
22    if (iVar2 != 0) {
23                    /*
24                       */
25      rVar3 = RF_Object_20002fd0.state.mode_state.cmdFs._12_4_;
26    }
27  }
28  else {
29                    /* get offset to pClient's callback
30                       */
31    rVar3 = findPClientOffset(pClient,pCb,0);
32  }
33  if (rVar3 == 0) goto LAB_0000cd96;
34  if ((*(byte *)(rVar3 + 0x2f) & 0x80) == 0) {
35 LAB_0000cd94:
36    uVar4 = 4;
37  }
38  else {
39                    /* if the  HashMatch pClient and startTrigger are the same, startTime will be
40                       equal to rVar3 */
41    if (RF_Object_20002fd0.state.mode_state.cmdFs._12_4_ == rVar3) {
42      *(byte *)(rVar3 + 0x2f) = (byte)(1 << (pCbFlags & 1)) | *(byte *)(rVar3 + 0x2f);
43                    /* len = 100 */
44      if ((pCbFlags & 1) == 0) {
45                    /* len = 80 */
46        pOp = 0x4010001;
47      }
48      else {
49        pOp = 0x4020001;
50      }
51      00013b34_RFCDoorbellSendTo(pOp);
52      if (iVar5 != 0) {
53        FUN_0000d6dc(pClient,RF_Object_20002fd0.state.mode_state.cmdFs.startTime,iVar5);
54      }
55    }
```

Figure 25: customRatFun Body

Analysis of Error_Check_StartTrigStartTime

One of the functions analyzed this semester was a FUN_0000d584 (later renamed to *Error_Check_StartTrigStartTime*) that was called multiple times in the shared function. The result of this function was used to conditionally set the RF_RatChannel mode between RAT Capture and RAT Compare.

```
byte Error_Check_StartTrigStartTime(int Ret_Array,uint8_t startTrigger,ratmr_t RF_obj_startTime)
{
```

Figure 26: Error_Check_StartTrigStartTime Method Signature

This function is compromised of 2 main conditionals each checking the validity of the passed in startTrigger (ID of the trigger that starts a Radio Operation Command) and startTime (Actual start time of the Radio Operation Command) fields of the Radio Operation Command.

```
15  uVar2 = (uint)startTrigger;
16  noErrorStartTrig = uVar2 == 0;
17  noErrorStartTime = RF_obj_startTime == 0;
18                    /* uVar2 = 20003008
19                       uVar2 + 0x28 = 20003030
20                       uVar2 + 0xc = 20003014 = RF_EventSync
21
22                       Ret_Array = **ratmr_t
23                       Ret_Array + 0x24 = "Ret_Array[6] = (ratmr_t *)0x0; "
24                       Ret_array + 0xc = "Ret_Array[2] = POTEN_TX_PKT; "  */
25  if (((uVar2 != 0) && (*(uint *)(uVar2 + 0x28) != 0)) && (*(uint *)(Ret_Array + 0x24) != 0)) {
26                    /* if PotenTXPKT event equals the uVar2 eventsync, specify uVar1 "offset" to 0
27                     */
28    if (*(undefined **)(Ret_Array + 0xc) == *(undefined **)(uVar2 + 0xc)) {
29      iVar1 = 0;
30    }
31    else {
32                    /* PTR_DAT_200030e0 is called by RF_done function, could be asking whether the
33                       RF_event is a RF_Done event, in this case set iVar1 to DAT_200030e8 */
34      iVar1 = DAT_200030ec;
35                    /* else set uVar1 to data specified in RF_init_maybe */
36      if (PTR_DAT_200030e0 == *(undefined **)(uVar2 + 0xc)) {
37        iVar1 = DAT_200030e8;
38      }
39    }
40                    /* offset calculation using addresses of retArray, uVar2, and uVar1 */
41    iVar1 = ((*(uint *)(Ret_Array + 0x24) >> 2) - (*(uint *)(uVar2 + 0x28) >> 2)) - iVar1;
42                    /* bounds checking, if error on offset calculation, return */
43    if (0x2fffffe < iVar1) {
44      return 0;
45    }
46    if ((-0x10000001 < iVar1) && (iVar1 < 1)) {
47      return 0;
48    }
49    noErrorStartTrig = true;
50  }
```

Figure 27: 1st Conditional

Initially on line 25, the validity of startTrigger ($uVar2 \; ! = \; 0$), checks the validity of RF_EventSync value ($uVar2{+}0x28! = 0$), and the validity of the POTEN_TX_PKT pointer ($Ret\_Array + 0xC \; ! = \; 0$). If all is valid we check if the POTEN_TX_PKT pointer is not equal to the RF_EventSync field, if this true then we pull the address of a 000001ec_read_radio_SRAM_GREEN function *(further analysis required for functionality of this function)* and use a bitwise arithmetic to create an "offset" using values of Ret_Array and startTrigger, after doing some bounds check on this offset we set a local boolean to true *(noErrorStartTrig = true)* Therefore we conclude that the 1st conditional is a validity check on startTrigger.

```
51                    /* RF_obj_startTime = 20003004
52                       RF_obj_startTime + 0x24 = 20003028 = "0001f9c6" in our dump
53                       Ret_Array + 0x28 = "Ret_Array[7] = (ratmr_t *)0x0; " */
54  if (((RF_obj_startTime != 0) && (*(uint *)(RF_obj_startTime + 0x24) != 0)) &&
55     (*(uint *)(Ret_Array + 0x28) != 0)) {
56    if (*(undefined **)(RF_obj_startTime + 0xc) == *(undefined **)(Ret_Array + 0xc)) {
57      iVar1 = 0;
58    }
59    else {
60      iVar1 = DAT_200030ec;
61      if (PTR_DAT_200030e0 == *(undefined **)(RF_obj_startTime + 0xc)) {
62        iVar1 = DAT_200030e8;
63      }
64    }
65    iVar1 = ((*(uint *)(RF_obj_startTime + 0x24) >> 2) - iVar1) - (*(uint *)(Ret_Array + 0x28) >> 2)
66    ;
67    if ((0x2fffffe < iVar1) || ((-0x10000001 < iVar1 && (iVar1 < 1)))) {
68      return 0;
69    }
70    noErrorStartTime = true;
71  }
72  return noErrorStartTrig & noErrorStartTime;
73}
```

Figure 28: 2nd Conditional

Similar to the 1st Conditional, we initially check the validity of startTrigger, RF_EventSync, and POTEN_TX_PKT pointer (line 54) as well as check if the POTEN_TX_PKT pointer is not equal to the RF_EventSync field. Where it differs is that we now calculate an "offset" using the values of startTime and Ret_Array and after some bounds check sets a local boolean to true *(noErrorStartTime = true)*. If both local booleans are raised then we return 1 (no error) and 0 if either are not raised (error).

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 29: 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 30: 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 31: 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 32: FUN_0000b6a0 lines 38-41

The `if` statement is checking if `iVar2` is populated with data, which the team suspects is a com-

plete 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 FUN_00007fc4 Function:
Additionally found in the references of the TX command structure, this function modifies startTime and startTrigger. This function relies heavily on the parameter passed in, with lots of if/else statements. Using Ghidra's tools, we determined that the parameter passed in is equal to 1. Using that, we can identify the control flow to see which conditionals will execute. This leads us to a function call of FUN_0000df28() which then either calls FUN_0000aae4() or FUN_0000b15c() which both seem to modify values of memory around Poten_TX_pPkt and modify startTrigger. In initial research of the FUN_0000b15c() function, the modification of values around Poten_TX_pPkt did not execute due to conditionals and research efforts were moved to FUN_0000aae4(). This function modifies startTrigger, setting the trigger type to TRIG_NOW. This aligns with the manual, in that the TX ADV structure needs the preTrigger to be TRIG_NOW for transmission to start. Further analysis is required to identify functionality.

Packet Processing
The 00002520_packet_processing function was also one of the functions of interest since it operates on packets in the RX queue. Understanding the behavior and logic of this function is crucial to understanding how the camera expects data packets to be, which will later help in creating malformed data packets. This function takes the current packet in the RX_queue and uses its header and payload bits to process the payload. The function is broken down into four main components: Error Checking, Setup, Processing, and Error Handling.
Error Checking: The first part of the function works on the inputs to the function in order to ensure that the radio is still properly transmitting and there have not been any fatal errors. These include checking the status of the radio, in order to ensure that the radio is still transmitting properly and is in the "OK" state.

```
102       if ((param_4 == 0 && ppuVar2 == (undefined **)&DAT_00010000) &&
103          (rfc_CMD_PROP_RX_ADV_s_20002378.status == 0x3400)) {
104 LAB_000025f8:
105         DAT_200024a2_status_not_ok = rfc_CMD_PROP_RX_ADV_s_20002378.status;
106         DAT_200024a0 = sVar7;
107         PTR_DAT_200024d0 = (undefined *)param_3_radio_status_check;
108         DAT_200024d4 = param_4;
```

Figure 33: An example of an error check. This portion is checking the status of the rfc_CMD_PROP_RX_ADV_s_2000237 through its status attribute.
There are many possible radio statuses available, and a complete list is seen in Figure 34:

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 34: A complete list of possible radio statuses.

Setup: After ensuring that the radio is still operating normally, the function continues to set up variables for the later sections. In this part of the function, the header provides information regarding the format of the packet, including the length of the packet, the length of the CRC, and whether or not whitening is enabled.

```
165     Packet_Second_Byte = queue_start_PTR_LOOP_20003a54[9];
166             /* Packet_First_Byte is pointing to 20003bf0 - which is the PktLength
167                                 Found from CurrEntry (20003be8 + 8) */
168     DAT_20003ae5 = 0;
169             /* Get the packet length from the first 11 bits of the packet header.
170                PTR_LOOP_20003a54 is start of loop and points to current data entry */
171     Total_Pkt_Len =
172          (ushort)(byte)queue_start_PTR_LOOP_20003a54[8] + (Packet_Second_Byte & 7) * 0x10
173             /* Checks if bit 12 of the header is 0 to find the length of the CRC of the
174                packet. Bit 12 is 0, so the CRC is 32 bits. (Header bits go from 0 to 15
175             /*
176                     total_length = transmit_len + CRC_LEN; /* CRC_LEN is 2 for CRC-
177                     16 and 4 for CRC-32 */ */
```

Figure 35: One portion of the setup section of this function. Here, the current packet in the RX_queue is being used to calculate the length of the packet and find and store the location of the second byte of the packet (lines 171 and 172).
Processing: After the variables are set, the function uses them to operate on the payload of the packet. The processing section is the main area of focus for this semester's research.
Within the processing section, there were a few subfunctions that the team investigated this semester. Each of these functions was determined to be important because they either edited the RX queue, or they edited the packets in the queue. The functions include FUN_00010fb8_queue_pop, FUN_0000f934_GREEN_edit_value, and FUN_00011684_CMP.
FUN_00010fb8_queue_pop
FUN_00010fb8_queue_pop is a function that is called during the processing section of the 00002520_packet_processing, among other places. This function sets the current entry in the queue to the next entry.

```
6  void FUN_00010fb8_queue_pop?(void)
7
8  {
9    if (DAT_20003a50_packet_first_byte != 0) {
10     FUN_00015064_reset_byte?();
11   }
12   (*(code *)PTR_disable_interrupts+1_200023ac)();
13   *(undefined *)(*(int *)Data Entry Queue_20003a60.pCurrEntry + 4) = 0;
14   Data Entry Queue_20003a60.pCurrEntry = *(undefined **)Data Entry Queue_20003a60.pCurrEntry;
15   (*(code *)PTR_enable_interrupts+1_200023b0)();
16   if ((DAT_20003ae3 != '\0') || (DAT_20003ae5 != '\0')) {
17     DAT_20003ae3 = '\0';
18     DAT_20003ae5 = '\0';
19     FUN_0000d36c_interrupt?();
20     if (DAT_20002916 != '\x01') {
21       FUN_0000525c_interesting(0);
22     }
23   }
24   return;
25 }
26
```

Figure 35: The `FUN_00010fb8_queue_pop` function.

In this function, the first byte of the current packet is cleared, if it was not already 0, which is seen in lines 9 and 10 of figure 34. Next, the pointer to the next entry in the queue is set to 0, which is seen on line 13 in Figure 34. Finally, the `pCurrEntry` of the `Data_Entry Queue_20003a60` queue is set to the current entry's next entry pointer, effectively popping the first entry from the queue, which occurs on line 14 in Figure 35.

0000f934_GREEN_edit_value

`0000f934_GREEN_edit_value` is also called during the processing section of the packet processing function. This function takes in three parameters, an address, a value, and a length, that are used later in the function. At a very high level, this function is using the parameters passed in to modify the value parameter passed in and edit the value that is stored in the passed in address. At the beginning of the function, there is an if statement that checks the if the address of the function is a multiple of four. Based on the results of this boolean, the flow of the function splits into one of two cases.

The first case occurs if the address passed into the function is not a multiple of four. This case is shown in Figure 36. The function then checks if the length passed in is 0, on line 26. If so, it decrements length and increments the value passed in, which occurs on lines 28 and 30 respectively. This will repeat until the value reaches a multiple of four. Once this condition is satisfied, it will check if the length is equal to 0, and if so, it will return the address passed into the function.

```
22   if (((uint)address & 3) != 0) {
23     do {
24       bVar5_length_bool = length == 0;
25                     /* if length is not = 0 */
26       if (!bVar5_length_bool) {
27         *(byte *)puVar3_value = value;
28         length = length - 1;
29         bVar5_length_bool = length == 0;
30         puVar3_value = (uint *)((int)puVar3_value + 1);
31       }
32                   /* What is this doing?
33                      Checks if last two bits are both 0 - multiple of 4?
34                      Does not necessarily decrement length all the way to 0
35                      checked later */
36       if (!bVar5_length_bool) {
37         bVar5_length_bool = ((uint)puVar3_value & 3) == 0;
38       }
39     } while (!bVar5_length_bool);
40     if (length == 0) {
41       return address;
42     }
43   }
```

Figure 36: The first case of the `0000f934_GREEN_edit_value` function.

The second case occurs if the address passed in is not a multiple of four. This case is show in Figure 37. In this case, the function concatenates the value passed in to itself and sets the result of this to the local value_dword variable, as seen on lines 44 and 46 respectively. Then, it will check the length parameter: if it is less than 15, the function will set the first four bytes in the local value variable to the value stored in value_dword. It will then decrement the value of the local variable uVar1 by 16, until it reaches a value that is less than 15. This functionality can be seen in Figure 37.

```
44   uVar1_value_concat = CONCAT11(value,value);
45                     /* dword = double word */
46   value_dword = (uint)uVar1_value_concat;
47   puVar1_modified_value = puVar3_value;
48   if (3 < length) {
49     value_dword = CONCAT22(uVar1_value_concat,uVar1_value_concat);
50     if (7 < length) {
51       if (0xf < length) {
52         uVar1 = length - 0xf;
53         length = length & 0xf;
54         do {
55           *puVar3_value = value_dword;
56           puVar3_value[1] = value_dword;
57           puVar3_value[2] = value_dword;
58           puVar3_value[3] = value_dword;
59           puVar3_value = puVar3_value + 4;
60           bVar5_length_bool = 0xf < uVar1;
61           uVar1 = uVar1 - 0x10;
62                     /* decrements uVar1 until is it < 0xf and sets puVar3_value to value_dword at
63                        the first 4 locations */
64         } while (bVar5_length_bool && uVar1 != 0);
65       }
```

Figure 37: The second case of the `0000f934_GREEN_edit_value` function.

Finally, this function will set the the fourth, third, second, and first bytes of the value variable to the local variable value_dword. The assignments happen in this order on lines 69, 77, 84, and 88 respectively in Figure 38. Finally, the function returns the address passed in.

```
                    /* Checking bit in 4th position */
      if ((length & 8) != 0) {
        *puVar3_value = value_dword;
        puVar3_value[1] = value_dword;
        puVar3_value = puVar3_value + 2;
      }
    }
                    /* Checking bit in 3rd position */
    puVar1_modified_value = puVar3_value;
    if ((length & 4) != 0) {
      puVar1_modified_value = puVar3_value + 1;
      *puVar3_value = value_dword;
    }
  }
                    /* Checking bit in 2nd position */
  puVar3_value = puVar1_modified_value;
  if ((length & 2) != 0) {
    puVar3_value = (uint *)((int)puVar1_modified_value + 2);
    *(short *)puVar1_modified_value = (short)value_dword;
  }
                    /* Checking bit in 1st position (msb) */
  if ((length & 1) != 0) {
    *(char *)puVar3_value = (char)value_dword;
  }
  return address;
}
```

Figure 38: The final part of the second case of the `0000f934_GREEN_edit_value` function.

FUN_00011684_CMP

`FUN_11684` performs a string compare and processes the packet that is passed to `FUN_11684` by comparing two variables. These two variables are data looped until they are zeroed out or the first one passed through becomes less than the second one. After that is done there is one final check done by the `FUN_0001128_check0_TX` function before they are returned.

```
void FUN_00011684_CMP(undefined4 *param_1_DAT_200026c4,undefined4 *param_2_DAT_200017e0)

{
  undefined4 uVar1;
  undefined4 *puVar2;
  undefined4 *puVar3_param2;
  undefined4 *puVar4_param2;

  *param_1_DAT_200026c4 = 0;
  if (DAT_20002916 != '\x01') {
    FUN_0000525c_interesting_error_check(1);
  }
  uVar1 = (*(code *)PTR_disable_interrupts+1_200023ac)();
  puVar3_param2 = (undefined4 *)*param_2_DAT_200017e0;
  puVar2 = param_2_DAT_200017e0;
  if (puVar3_param2 != (undefined4 *)0x0) {
    puVar4_param2 = puVar3_param2;
    do {
                    /* comparing params to each other
                       Thi loops until param2 is 0 or greater than param 1 */
      puVar3_param2 = puVar4_param2;
      if ((int)param_1_DAT_200026c4[1] < (int)puVar4_param2[1]) break;
      puVar3_param2 = (undefined4 *)*puVar4_param2;
      puVar2 = puVar4_param2;
      puVar4_param2 = puVar3_param2;
    } while (puVar3_param2 != (undefined4 *)0x0);
  }
  *puVar2 = param_1_DAT_200026c4;
  param_2_DAT_200017e0 = (undefined4 *)*param_2_DAT_200017e0;
  *param_1_DAT_200026c4 = puVar3_param2;
  if (param_1_DAT_200026c4 == param_2_DAT_200017e0) {
    FUN_000112a8_check0_TX();
  }
```

Figure 39: The FUN_00011684_CMP function.

Error Handling: This final portion of the function contains error handlers. If there were any errors that occurred in the Error Checking portion of the function or while the packet was processing, there will be a GOTO label that matches one of the handlers in this section. This section accounts for handling any unexpected behaviors.

6 Conclusion

The purpose of this research is to gain a better understanding of the way that the Wyze Camera expects to receive, transmit, and process data. By reverse engineering the relevant parts of the disassembled code, we can gain more information about how the camera manages and transforms data. By gaining a better understanding of this, we will be able to send purposefully malformed data to the camera in order to see how the camera reacts. Sending malformed data will potentially result in the camera throwing errors or exceptions, or by exhibiting unpredictable behaviors. By monitoring how the camera reacts to these inputs, we may be able to find vulnerabilities that can be exploited in the camera. Having an idea of how the camera should behave in the ideal circumstance will also provide a baseline that we can use to compare to the camera's behavior with malformed data.

A future goal of the project is to complete this process automatically through a process called fuzzing. When fuzzing is performed on the camera, malformed data will be continuously sent to the camera, and the camera's output will be constantly monitored in order to look for unpredictable behavior, exceptions, or errors. This process can reveal information about the camera's vulnerabilities in a way that is faster and more efficient than manually reverse engineering the camera.

The end goal of this project is to take the information learned from reverse engineering and comparing it to the results gained from later fuzzing the code. This will show if fizzing is a viable alternative to reverse engineering when finding vulnerabilities in an IoT device, which will greatly increase the efficiency of this process.

7 References

[1] "Wyze," Wyze. https://www.wyze.com/

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

[3] 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

[4] Ahmed, D. (2024, February 20). Wyze cameras glitch: 13,000 users saw footage from others' homes. Hackread. https://www.hackread.com/wyze-cameras-glitch-users-saw-home-footage/