École Polytechnique Fédérale de Lausanne

Security of Modern Consumer-grade Internet-of-Things devices

by David Kleymann

Master Project Report

Prof. Dr. sc. ETH Mathias Payer
Thesis Advisor

Luca Di Bartolomeo
Thesis Supervisor

EPFL IC IINFCOM HEXHIVE
BC 160 (Bâtiment BC)
Station 14
CH-1015 Lausanne

January 5, 2024

Abstract

The goal of this semester project is to investigate the current state of software security in IoT consumer products by analyzing a selection of popular devices on the occasion of the Pwn2Own 2023 Toronto competition. The goal of the Pwn2Own 2023 Toronto competition is to find vulnerabilities and achieve remote code execution on consumer products. We document our methodology and findings and give insights into what IoT software looks like and what attack surfaces devices are exposed to.

Contents

Chapter 1

Introduction

Ubiquitous computers that fall into the category of the Internet of Things (IoT) are a reality that is here to stay: The number of IoT devices in the world is growing and so are concerns about security and privacy [10]. Through smart light bulbs [9], security cameras [1], TVs [2], coffee machines [7], electric vehicle chargers [8] and more, customers and businesses are facing privacy and security threats from inside their own buildings and local networks. Apart from these Internet-enabled everyday appliances, there is also a similar trend for computing devices used for critical infrastructure, industrial control systems and medical devices to make communication over general-purpose wireless networks like Wi-Fi and Bluetooth the default, often making devices, originally designed to be robust and simple in operation, insecure in novel and unforseen ways. As Mirai botnets have demonstrated since 2016, insecure IoT devices are not only a problem for their owners and those who use them, but they pose a risk to the internet and our digital infrastructure. Due to the sheer number of IoT devices with network access, botnets like Mirai can achieve magnitudes more Distributed Denial of Service (DDoS) bandwidth then other botnets. The potential reasons for low security in so many IoT devices are different. Many consumer-grade IoT products are not aiming for a high-end market: Customers generally expect IoT devices to be cheap, since many types of IoT devices have relatively simple purposes, for example a camera should record video, a thermostat should regulate temperature. Compare this to desktop computers, which need to be fit for multiple purposes, say web browsing, 3D computer graphics, multi-tasking of numerous applications and virtually anything the user might want to do with it. Vendors of IoT devices often rush to bring new features to market, that can be immature in terms of security and robustness. Software is often pieced together uncleanly from legacy parts of older products and completely different new implementations, leading to an incoherent patchwork of software components inside device firmware. It therefore requires knowledge of each of these parts to thoroughly assess security and even makes it hard for the vendor to secure and patch the device. Many times vulnerabilities of open-source software running on IoT devices, that are fixed in the most commonly used versions, need to be patched in the older or custom versions the vendors use. This is also in part due to the

System-on-Chip (SoC) and Chipset manufacturers quickly retiring product support, providing bad Operating System (OS) support (e.g. device supports only old Linux version X.X), and forcing the IoT device vendor to use this old system and toolchain, or spending time and effort to port device-specific details to a more recent system. Furthermore, IoT devices that need to be cost-effective are also very low-powered computing devices, that do not benefit the increasing complexity of mainstream OSes like Windows and Linux. Many of the devices we researched have only a few MB of Random-access memory (RAM): The router has 256 MB and the Wyze camera has just 128 MB. Thus it could also be a performance decision to use an older OS instead of a recent version. Despite all of these difficulties, it is not impossible to build secure IoT products. Vendors need to start rewriting their software with security and maintainability in mind.

The company that sells one of the devices we selected for this project, Wyze Labs, has a track record of handling vulnerabilities in their products incorrectly. One of the devices we consider in this work, the Wyze Cam v3, is not a new product. First released in 2020, it is an incremental improvement from the previous versions v2 and v1 and runs similar software. This also becomes clear when researching previous vulnerabilities in these cameras. In 2022, Bitdefender published three vulnerabilities found originally in 2019 for all Wyze Cam versions, including remote code execution and access to the local SD card storage [3]. Wyze failed to disclose these vulnerabilities for 3 years.

"We would be remiss not to note that Wyze Cam Pan v2 was one of the cameras that had a security flaw that the company took a long time to acknowledge publicly and only finished patching last year (except for the Wyze Cam v1, which it could not patch because of its hardware limitations). [...] Wyze's mistake here was to not talk about this one for years. I'd like to think that Wyze has learned from this and has since strengthened its approach to security (and its app supports two-factor authentication), but it's something worth keeping in mind." [12]

The new vulnerabilities of this device found in this semester project, in addition to those reported as a result of the Pwn2Own 2023 competition show that Wyze Labs has not improved the security of their devices enough. Furthermore, their response to the previous issues shows that with insufficient expertise or manufacturer support to customize the OS in the firmware, securing low-end IoT devices a posteriori is challenging: "Unfortunately, despite extensive efforts stretching into 2022, we found Wyze Cam v1 (last sold in March 2018) could not support the necessary security updates. The limited camera memory that prompted us to create Wyze Cam v2 directly prevented patching these issues on that product." [14]

Chapter 2

Background

Competition rules (Pwn2Own 2023 Toronto)

Since the Pwn2Own 2023 Toronto competition is the inspiration for this semester project, we will describe the background and rules of the competition. Pwn2Own is a series of competitions started in 2007. The general goal of the competitions is the exploitation of unpatched vulnerabilities in devices or software products, updated to the most recent release. Achieving the goal yields a monetary prize and the targeted device itself, hence the name (*Pwn* [= exploit] to *own* the device). The competition is organized by Trend Micro, a software and server security company and the Zero Day Initiative, an organization paying money in exchange for unreported vulnerabilities to report them to vendors and get them patched. The contest is open to any person of full age not from a US embargoed or sanctioned state. The 2023 Toronto edition of the contest had multiple categories of devices (Mobile phones, Google devices, Smart speakers and more). We focus on the Small Office / Home Office (SOHO) Smash-up category of devices, because many these devices were the easiest to obtain physically, with the exact hardware and software version required for the competition. The specific security violation goals for each device can be different. We selected a router from the list of allowed routers for the SOHO Smash-up category and a Wi-Fi-enabled surveillance camera from the Surveillance Systems Category. An exploit for remote code execution on the camera alone would be rewarded with up to 30000 USD. To win any prize, the exploit needs to achieve the goal on the competition sponsor's device, which is fully patched to the latest software version on the day of the competition. Participants need to demonstrate in one of three attempts of 10 minutes each the success of their attack on the target. Furthermore, the rules state that *every attempt must achieve the goal without any user interaction*, without requiring the legitimate user to reboot or log on or off. Vulnerabilities exploited need to be fresh, i.e. unknown to the vendor and the public and each found vulnerability can only be used for one device. For entries requiring a Man-in-the-Middle attack, the competition sponsor can choose to accept or deny it with a smaller prize.

```
0059f788 3c 00 b1 8f     lw          s1,local_24(sp)
0059f78c 38 00 b0 8f     lw          s0,local_28(sp)
0059f790 08 00 e0 03     jr          ra
0059f794 60 00 bd 27     _addiu      sp,sp,0x60
```

Figure 2.1: A MIPS delay slot instruction following a return

Devices selected and target architecture

We selected the Wyze Cam v3 and the TP-Link Omada VPN Router ER605 V2, since these were the most interesting devices, with firmware downloadable without requiring the purchase of the actual devices. They are also cheap to obtain online, the camera was available for just 33 CHF and the router cost 58 CHF and could be shipped from Switzerland. Both are based on MIPS32 revision 1 CPUs (little-endian, also known as mipsel). Their 32-bit Restricted Instruction Set Computer (RISC) architecture is historically very popular for use in low-power networked devices. This architecture has a branch delay slot after every control flow instruction, that will be executed before the branch takes effect. In Figure 2.1, the stack space of a function is freed just before the function returns execution to the caller. Loads and stores use a base from a register and an signed 16-bit offset.

Ghidra, a powerful decompiler, decompiles most encountered binaries correctly, but sometimes fails to recognize references to strings due to this addressing mode. Certain libraries also required some tweaking of the default settings for MIPS decompilation to stop Ghidra from ignoring references to low memory addresses common in RISC binaries. Alternative decompilers we tried, like Binary Ninja and IDA Pro had their own issues with decompiling binary code found in the firmware of both devices and we decided to work with Ghidra for most of our reverse engineering tasks, since we had experience writing scripts for it.

Chapter 3

Analysis of selected devices

We now take a closer look at each of the devices tested and provide the procedure we followed to unpack the vendor's firmware images, extract filesystems and analyze binaries. Furthermore, we present our ways of adding priviledged access for our debugging purposes in the devices, from using existing backdoors to booting custom OS images running reverse shells in the background. We also discuss potential future approaches that we only explored partially. In our research, we prioritized the Wi-Fi camera, since it was a more promising target and also had a reward for successful exploitation, while an exploit on the router alone would not win any prize.

Wyze Cam v3

The Wyze Cam v3 seems to be a popular wireless security camera on the market, with many positive reviews. Although it is over 3 years old, it provides a growing set of features: Motion detection, AI-powered person detection, cloud video footage storage and alerts sent to your phone app. It features 2.4 GHz band Wi-Fi, a speaker and microphone and a 1080p image sensor. Opening up the device, we see that it runs on an Ingenic T31 SoC that includes an XBurst 1 CPU, 512 MB of RAM and an audio codec, image and video processor and an additional RISC-V core. The Ingenic T31 seems to be a widely used platform for wireless security cameras. InnoPhase offers their "Smart Video ADK" (see Figure 3.1) development platform, whose core board construction looks similar to the insides of the Wyze Cam v3.

We believe this example also led to the development of the following very similar consumer products in the Wi-Fi camera categories using the Ingenic T31: The Eufy 2K, the Atomcam and the Azarton-C camera. We assume that currently there may be a common Original equipment manufacturer (OEM) producing these devices, that are branded by Chinese and American companies. This means that any vulnerabilities present in the Wyze Cam v3 may also apply to many other Wi-Fi

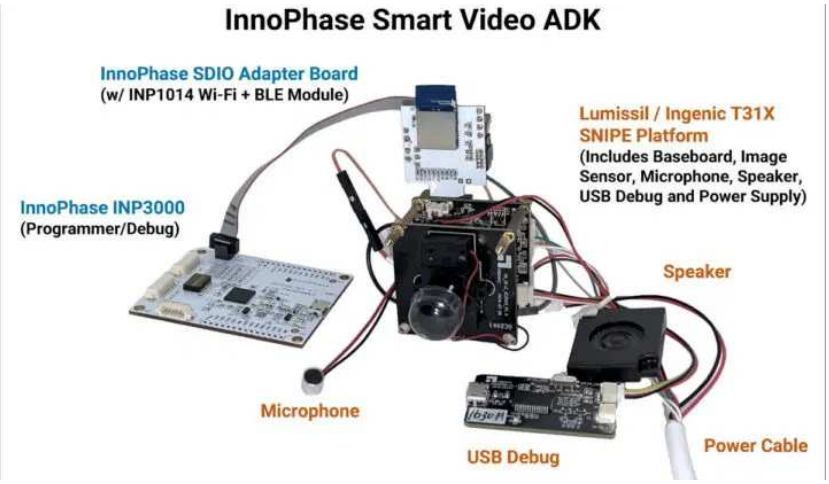

Figure 3.1: InnoPhase Smart Video ADK

smart cameras. One of the main binary files in the firmware is called "iCamera", a name of an older popular Wi-Fi camera, and many configuration files and code reference "Hualai", a company name seemingly present everywhere in other Wi-Fi camera firmwares and hardware, leads us to believe that the software running on these cameras might be very similar too. Upon disassembly of our ordered Wyze Cam v3, we found what we presumed to be an Universal asynchronous receiver-transmitter (UART) port on the side. The 3 pin holes were the largest unused contact or test points on the Printed circuit board (PCB) of the camera and checking the T31 chip datasheet and using a Multimeter to test for continuity between pins of the QFN88 chip package and the pin holes, we confirmed RX, TX and ground pins for the serial port. It is enabled and usable by default, also running at a baudrate of 115200. The datasheet was also of help for finding the pin for the chip reset signal, useful in case we need to automatically reboot the chip, and JTAG debugging ports. After soldering everything we deemed to be useful we protect the fragile microsolder-points with hot glue (see Figure 3.2).

On the serial port, the camera gives us its full boot log output and a login shell. Unfortunately, we do not know the standard root password and we were unable to crack the salted password hash we later found inside the firmware. However, the boot log contains a lot of useful information, that might otherwise be harder to obtain from the firmware. We learn that again we have a U-Boot bootloader (very old version, released 2013 and compiled 2021), that mounts the SD memory card of the device and looks for a file called `factory_t31_ZMC6tiIDQN` on the first FAT32 partition. The OS booted is Linux 3.10.14, a version from 2016 by Ingenic compiled in 2022 according to compiled-in version strings. This gives us an idea of how old the whole toolchain used is, using gcc version 4.7.2 and hardware drivers from the same era. We learn that only the MIPS core of the CPU is actually booted, the SPI flash contains the firmware plus firmware backup and user configuration partitions and the camera uses the 'RTL871X' driver with the exact version 'v4.3.24.7_21113.20170208.nova.1.02' for the Rtl8189ftv Wi-Fi module.

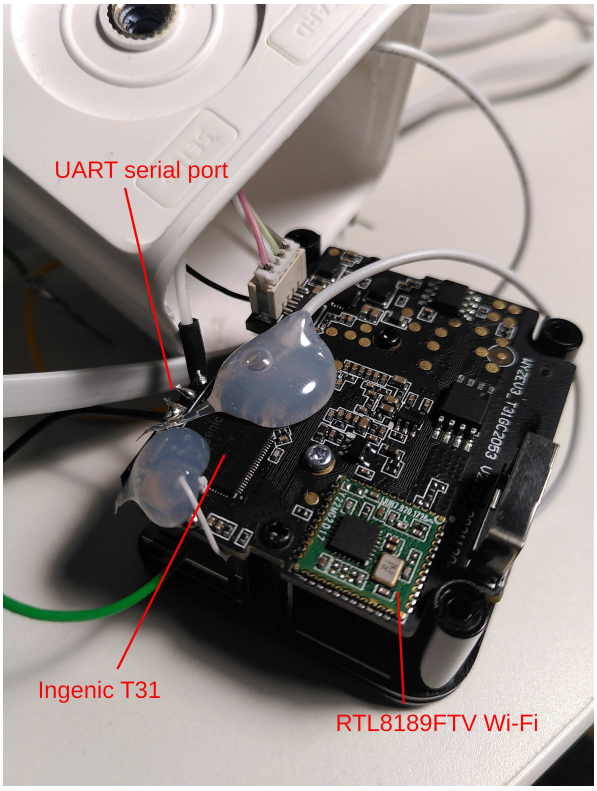

Figure 3.2: Wyze Cam v3, opened and with wires connected for debugging

Firmware analysis

Before we proceed by rooting the device, i.e. obtaining a local root shell for listing running processes, network connections and debugging programs on the device itself, we extract the original firmware images available for download on the vendors page. Binwalk shows the following contents of the firmware image called `demo_wcv3.bin`:

Listing 3.1:]Binwalk output on camera firmware image, omissions denoted by [...]

```
DECIMAL       HEXADECIMAL     DESCRIPTION
--------------------------------------------------------------------------------
0             0x0             uImage header, header size: 64 bytes, header [...]
    OS: Linux, CPU: MIPS, image type: Firmware Image, compression type: none,
    image name: "jz_fw"
64            0x40            uImage header, header size: 64 bytes, header [...]
    OS: Linux, CPU: MIPS, image type: OS Kernel Image, compression type: lzma,
    image name: "Linux-3.10.14__isvp_swan_1.0__"
128           0x80            LZMA compressed data, properties: 0x5D, [...]
2031680       0x1F0040        Squashfs filesystem, little endian, [...]
6029376       0x5C0040        Squashfs filesystem, little endian, [...]
```

We find two uImage headers, one for the entire firmware image and one for the Linux kernel image itself. The firmware also contains two different SquashFS filesystems, the first one being the root filesystem, the second one is mounted under `/system` on the camera and contains almost all the custom and Wyze-specific components including: Audio, Video, Wi-Fi and SD card drivers, numerous camera recording, firmware upgrade and cloud communication binaries, custom Wyze and open-source libraries and default configuration files. The directory `/system/bin` contains custom programs running as root on the camera, including 'assis', 'iCamera', 'iotclient' and 'timesync'. All of these binaries use no common protections, no Stack Canaries, they do not use NX/XI bit to separate executable code from data and stack or Address Space Layout Randomization. Only the root user exists on the device. We used Ghidra to reverse engineer some of the programs in the firmware specific to the Wyze camera and found heavy usage of code that runs commands in the shell from these binaries, usually without input sanitization. These commands passed to functions like `exec_shell_sync` or `exec_shell_sync2` pass through an internal message queue and then are executed with `/bin/sh sh -c` and the supplied command as a last argument in a forked child process. Commands are often constructed using standard string formatting functions like `snprintf` with `'%s'`. In other cases, even in the same binary, the entire command string is crafted directly and passed to `system` without any queuing. The inputs to the string formatting come from settings and values controlled by commands received over responses to HTTPS connections to Wyze cloud servers. Since they use `curl_easy_init` with a modern version of libcurl (v. 4.7.0 to 4.8.0) and safe settings, the HTTPS channels are considered to be secure and we can not trigger Command Injection vulnerabilities that rely on attacker control of these messages.

One script that was useful for reverse engineering programs of both the router and the camera is the automatic renaming of functions to their actual function name before compilation from log message print calls. Many functions would call a single standard logging function like this:

```
print_module_log_print("[ICAMERA]",4,"init.c","init_sdk_speaker",0x1e7,
                       "err: local_sdk_speaker_init fail...\n");
```

While most binaries were stripped of symbol names for the firmware of the Wyze Cam v3, the custom library containing this logging function had a meaningful name for the logging function and a Ghidra script in python can simply look for references to this function and extract the caller function name as a string from the arguments (`init_sdk_speaker`). For the TP-Link router, the address of a similar logging function was also easily determined. Function names obtained this way were very useful for reverse engineering, but many other functions without any log print calls needed to be analyzed manually.

Root access to device

While we found a vulnerability in the camera's set up procedure (see next chapter), we needed a simpler and preferably persistent way of accessing a root shell on the camera. Running the `passwd` command on the device or changing the root password hash in another way does not work: The entire file system is immutable except for the SD card's FAT32 partition, which is mounted on `/media/mmc`. We did however often use this partition to transfer custom kernel modules or shell scripts to the device. Instead, we decided to investigate the `factory_t31_ZMC6tiIDQN` that the bootloader was looking for during very early startup. A public GitHub issue on rooting the Azarton camera suggested that if this file should contain a uImage OS kernel image, U-boot would boot this kernel instead of the default one stored in the flash memory of the device. We confirmed that this was indeed working, by extracting the original kernel image from the downloaded firmware image and saving it under this filename on the FAT32 partition of our SD card. The next step was to change the default kernel command-line of the kernel image to include:

```
root=/dev/mmcblk0p2 rw rootwait rootdelay=5
```

This instructs Linux to mount the root file system from the second partition of the SD card, instead of the default `/dev/mtdblock2` partition of the internal flash storage and wait for the SD card to get mounted before attempting to mount the root directory. Initially, we did this with an SD card containing a SquashFS partition after the FAT32 partition (required to store the kernel image we boot). This SquashFS can easily be copied from the firmware image using the `dd` command, since the firmware image is not compressed, then the files inside the SquashFS can be copied to a temporary location on our machine. After changing the root password hash in our temporary location, we used `mksquashfs` to again obtain an immutable SquashFS image from this altered root file system, which can then be copied directly to `/dev/mmcblk0p2` (the second SD card partition).

The Linux kernel used is very specialized and many features are not enabled, therefore it took us some time to find a more flexible solution: Instead of SquashFS on the SD card, we now prepare a card with FAT32 in the first partition, and a small second ext3 partition. The Linux kernel supports mounting this ext3 partition from the SD card, allowing us to have a mutable root file system which we can also easily open and inspect on a different Linux computer by plugging in the microSD card into an SD-card reader. Now we could simply log in to the default shell on the camera using the UART serial port and our own root password. From this we started writing some reverse shells, which are scripts or programs that give us simple access to the internal shell of the device from the internet via a public server we ran temporarily. This was very beneficial, since it allowed us to work on different components of the Wyze camera in parallel.

We also found a way to compile a completely custom Linux kernel to boot on the camera, thanks to the toolchain and so-called "Linux SDK" from Ingenic on GitHub ("unofficial-ingenic-t31"). These repositories contain a gcc compiler version 4.7.2 for MIPS, the exact same version Wyze used to compile their Kernel image, as well as the exact Linux version 3.10.14 source code. Most of the default build configuration options seem to look correct for the 'isvp' platform and with some alterations to them we were able to build a kernel uImage that successfully boots the camera when placed in the `factory_t31_ZMC6tiIDQN` location of the SD card. This was later useful for exploring the option of kernel debugging, but we did not use this custom kernel by default, because we could not predict potential behavioural differences of our kernel and the kernel used in Wyze's firmware images. These differences may be subtle, but could lead to different memory layouts, kernel and driver behaviour and might prevent us from finding crashes or vulnerabilities otherwise present in the stock firmware. Despite this, this kernel source code proved useful for compiling our own version of the wireless interface driver for more debugging output, as well as compiling other custom kernel modules used for debugging. Kernel modules can be stored on the SD card's FAT32 partition and loaded using `insmod` from inside the patched original kernel.

Debugging and introspection

Even with root access and a shell on the device, we found that there are almost no debugging tools we could use on the camera. This is an issue we also encountered with the Omada router, because reverse engineering tools like strace and gdb were missing from the system. We had partial success with emulating binaries from the Wyze Cam v3 firmware using QEMU, but debugging multi-threaded programs and emulating Wi-Fi hardware and General-purpose IO (GPIO) accessess proved to be too much of a challenge to deal with. Our next move was trying to get all tools we needed, most importantly gdb or gdb-server and strace. Because cross-compiling these programs with the old gcc version 4.7.2 toolchain and both libc and uclibc would have meant multiple days of work, we instead searched for statically compiled binaries for these tools. Our own attempts at statically compiling gdb failed to execute properly on the camera. We were able to find some working statically compiled binaries of strace [11], GDB and socat (used for reverse shells we used for remote debugging). While

strace worked as expected, it slowed down the execution on the single-core CPU so much that most processes on the camera did not run smoothly anymore. Performance of the programs running on the Wyze Cam v3 was fragile with little room for our own processes. GDB was less straightforward, because it executed but reported errors and warnings. The most important issue was that GDB was unable to set and stop at breakpoints correctly for multiple threads. Apart from this, GDB would regularly crash and cause a reboot. This left us with no effective way of debugging multi-threaded programs on the camera, which most of the interesting and custom processes were.

With our developed debugging and introspection capabilities on the running camera, we can further explore the system. The Wyze Cam v3 is running a very lightweight Linux setup, with about 35 regular processes running. To compensate for only having a single core, but needing to respond to camera and network data, some processes have a large number of threads: The 'iCamera' process has around 71 threads in idle operation, meaning that no streaming client is connected. We used Ghidra to reverse engineer the operation of the processes running on the camera. As previously metioned, almost all processes use HTTPS to connect to servers on the internet. One such example is the use of AWS IoT MQTT services to send telemetry data to the cloud and query for firmware upgrades. It is unclear why, even though automatic firmware updates seem to be implemented, we did not receive any during our time running the camera with downgraded firmwares and access to the internet. The camera did however download a tar archive, via HTTPS, containing binary programs and shared library files seemingly belonging to something called 'wyzeedgeai' (Wyze Edge AI). We concluded this refers to the AI person and object detection advertised by Wyze, but it probably did not fit into the firmware image. We did not investigate it much further since it only came to our attention at a later stage during the project, but these binaries also run as root and make network connections, still through HTTPS. One service of the 'iCamera' binary that does not communicate via HTTPS is called TUTK. TUTK is a very obscure protocol wrapping a DTLS session and listening on the UDP port number 32761 on the Wyze Cam v3. The protocol is developed by Throughtek, who advertise it for use with IP cameras and child monitor cameras. We expect the code handling the TUTK protocol is also shared across many IoT cameras, since we also found a category of settings named "TUYA" in the configuration files of 'iCamera' (under `/config/.user_config`), which is a reference to a Wi-Fi child camera called "Tuya Tutk". While we were not able to reverse-engineer the TUTK protocol in its entirety, another competition participant later revealed to have found a debugging backdoor or authentication workaround allowing them to establish a TUTK session with the camera [13]. From this, they were able to exploit a "stack buffer overflow in JSON unpacking" [13]. Unfortunately, we expect there to be many such trivial vulnerabilities in the software of the Wyze Cam v3, some of which we found and document in the next section. This is due to the sheer size of the binary (3.7 MB for 'iCamera' alone), the way that many different sources of legacy code for different functions are used in uncontained and insecure fashion. For example, the iCamera binary seems to call functions from multiple different JSON libraries, some of which are statically compiled while others are shared libraries, making code maintenance complicated for the software vendor. Furthermore, this binary contains numerous occasion where user input or strings received over the network are included into format strings (using "%s") and executed in the shell.

Kernel modules

Since TLS made high-level programs of the camera unexploitable to us, we decided to assess the security of the custom kernel modules of the Wyze Cam v3. The list of kernel modules loaded is obtained by running `lsmod` on the camera and we can see that it includes: Wi-Fi module driver (rtl8189ftv), Camera driver ('sensor_gc2053_t31'), Audio driver ('audio') and Image Signal Processor driver ('tx_isp_t31'). Most of these are assumed not to be exploitable for our objectives, since the drivers will unlikely work with attacker-controlled input directly, but rather process some video input or audio signal input. The Wi-Fi driver however is of great interest: The Wi-Fi is the camera's only method of communication, so it should always be loaded and working in normal operation. As we saw earlier from reading the camera's boot log, the rtl8189ftv kernel module seems to be a version from early 2017, meaning it may be poorly maintained and potentially vulnerable. While we did not have access to the exact version of the Realtek Wi-Fi driver that Wyze used, we tried to find the source code of the closest available version. The only version that ended up working when compiled and loaded onto the camera is the slightly older and different rtl8189fs 'v.4.3.24.4_18988.20160812'. By setting the Makefile to compile the driver to look for the unsupported rtl8189ftv chip and treat it as a different one, we could compile a working kernel module using the gcc 4.7.2 MIPS compilation toolchain and kernel previously mentioned. Loading it instead of the original rtl8189ftv driver results in much more debugging output from the Wi-Fi kernel, as we enabled in the source code we found, at a cost of significantly less stable Wi-Fi operation. Adding our own `printk` function calls inside the driver also helped immensely in debugging crashes we later found in the wireless driver.

Listing 3.2: Self-compiled wireless driver starting up

```
[root@WCV3:configs]# insmod /media/mmc/8189fs.ko
[15:54:11] [ 6220.695456] RTL871X: module init start
[15:54:11] [ 6220.701360] RTL871X: rtl8189fs v4.3.24.5_19847.20161102
[15:54:11] [ 6220.721800] RTL871X: build time: Oct 19 2023 12:07:00
[15:54:11] [ 6220.730396] wlan power on
[15:54:11] [ 6220.752609] RTL871X: CHIP TYPE: RTL8188F
[15:54:11] [ 6220.769110] RTL871X: rtw_hal_config_rftype RF_Type is [...]
[15:54:11] [ 6220.778510] RTL871X: Chip Version Info: CHIP_8188F_[...]
```

We tested the Wi-Fi driver's security in two main ways: Our first approach was to read the source code of the older version of the Realtek driver we found. The driver contains many different functions, many of which were disabled on the driver used on the Wyze Cam v3. We therefore identified crucial components of the driver that were present on the original driver (confirmed using disassembly of the original kernel module) and that we deemed to be closest to attacker-controlled data from received Wi-Fi packets. When initially trying to test the extent of our control of a certain length value in the drivers `rtw_security.c` file, we wrote a Scapy script to send a raw 802.11 frame to the camera's wireless card. This quickly led us to a crash caused consistently by small frames (e.g. 7 bytes of data). We will describe the details of this driver bug in the next section and continue by

showing how we subsequently improved our over-the-air fuzzing scripts with the goal of finding different crashes. Starting from a very basic script in the beginning, we kept the main concept of sending raw Scapy `Dot11` frames, with randomized type and subtype fields, the sender address (addr2 and addr3) is the MAC of the legitimate Wi-Fi AP that the camera is connected to in the meantime, and the destination (addr1) is the Wi-Fi MAC of the camera's wireless module. Payload of this `Dot11` frame are random bytes of random length (constrained by MTU). Since this script was sending Wi-Fi packets very quickly and we needed a way of tracing back crashes to the packet that caused it, we also save each packet we send out with a timestamp on the attacker's machine. This approach would often find the same crash we already had before, but did not cause any new crashes. Sending raw 802.11 frames using Scapy required a wireless interface in Monitor mode. We can put a wireless interface in Monitor mode using the `airmon-ng` of the airodump-ng suite.

Black-box over-the-air Wi-Fi fuzzing

Since this approach had some success so far, and we did not have a better way of black-box fuzzing the Wi-Fi driver, we thought about ways of improving it. Due to the inconvenience of using our own personal computers to send fuzzing Wi-Fi packets and constantly crashing or disappearing wireless access points, also being inspired by Marc Egli's Over-the-air LTE fuzzing techniques [4], we started building a Raspberry Pi-based over-the-air Wi-Fi fuzzer. It consists of the Raspberry Pi 3B single-board computer, an SSD attached over USB 3.0 for storing logging data and packets and two wireless interfaces. The Raspberry Pi was also connected to the Wyze Cam v3 via an USB to UART serial converter. Power to the camera is also supplied from the Raspberry Pi and we connect the Reset pin of the camera to a GPIO pin on the Raspberry Pi. This allows us to forcibly reboot the Camera automatically at any time. The camera's Wi-Fi module either connects to some external AP or to an AP run by the Raspberry Pi. To fuzz the camera's Wi-Fi driver, we let the camera reboot, check that it booted correctly and connects to the Wi-Fi AP, then send generated 802.11 frames in a loop until we detect signs of a kernel stack trace or crash report on the serial output of the device.

Unfortunately, this approach has some drawbacks. Since we send a lot of invalid Wi-Fi packets to the camera, we do not expect or check for acknowledgement packets. This means that unless we add debugging `printk` calls in the driver to dump incoming packets and parse this in the serial output, we have no feedback mechanism to check if our packet was actually received. Since the camera and the Raspberry Pi are physically close, we expected most packets to arrive, but we had to be careful not to send packets too fast, to not overwhelm our sending wireless interface. Additionally, we had more issues on the Raspberry Pi than before with sending any Wi-Fi packets using the same technique as before. This seems to be due to the wireless USB dongles we used, which would either silently discard all outbound packets in monitor mode , or did not properly support monitor mode.

Another option we did not test fully is to fuzz the Realtek driver using emulation with QEMU and supplying it Wi-Fi packets directly. We would start from the Wi-Fi driver source code, at least

the older version we were able to find, and patch it or hook some points in the code, supplying it with `sk_buff` structures generated by our fuzzer and emulating all hardware device interactions required to further process an incoming packet. On the other hand, this substantial effort might not pay off in the end and find crashes that are impossible to reach, because e.g. the Wi-Fi chip filters out certain invalid packets in the hardware itself.

To determine the exploitability of the bug found in the Realtek Wi-Fi driver, we required more powerful ways of debugging the kernel module than debugging the driver by inserting function calls to `printk` and observing the serial console output. Since we cannot use the Wi-Fi driver on a QEMU emulation system, we need some form of on-device debugging. KGDB with KDB is a possibility for this but requires us to compile our custom kernel, which is suboptimal for creating an exploit targeted towards the slightly different stock kernel. We therefore attempted to use the JTAG interface of the T31 chip with a J-Link debugging adapter. For this, we soldered thin wire to the reset pin 'PPRST_' , TCK and TMS pins, or components connected to them on the PCB. TDI and TDO can be connected using the serial port pin headers, since they use the same chip pin as the UART serial port RX and TX signals. Unfortunately, our attempts at JTAG debugging were not successful. We were able to reset the CPU, which proved useful for the automatic Raspberry Pi Wi-Fi fuzzing setup. We could not find the required JTAG parameters to set up openocd, most importantly clock speed and the IRLen of the T31. We assumed the CPU was the only device in the JTAG chain on the camera. We also tried to use the urjtag JTAG scanning tool, to no avail. We also suspect that JTAG debugging may have been disabled during manufacturing of the device. The datasheet of the Ingenic T31 documents the presence of an eFuse mechanism, a way to permanently program some settings of a chip. The datasheet does not mention how the fuses are programmed or what they control, but it is common to have a way of disabling testing and debugging functions for end-user products using the eFuse on a device.

We noticed that the bug found is present in many different versions of the Realtek Wi-Fi driver and is not specific to the MIPS architecture either. Since we can not emulate the Wi-Fi module but we can also not easily remove it from the PCB of the camera and probe it, we opted to buy a different, new Wi-Fi module that would be compatible with a similarly old version of the Realtek driver. We could use this to analyze the bug on a different, more debugging-friendly platform than the very restricted one the camera had. We bought the edimax EW-7811UN V2 USB Wi-Fi dongle containing the RTL8188EU chip and were able to use it on our personal x86_64 computers with a fully working gdb debugger. However, after running the Realtek driver on our machine, we learned that this newer generation Wi-Fi module always used hardware AES encryption and decryption, which bypassed the bug we were investigating.

TP-Link Omada VPN router

The TP-Link Omada ER605 V2 is a Gigabit-speed Ethernet router from TP-Link's Omada Software-defined Networking (SDN) line. It is advertised with features such as "Zero-Touch Provisioning[] Centralized Cloud Management, and Intelligent Monitoring" [6].The cloud management capabilities are specifically "Cloud access and Omada app for convenience and easy management" [6]. This is what initially drew our interest to the device, since this hinted at a large attack surface exposed to the internet. Further features also seem to have potential to be vulnerable: The router supports a few different VPN and WAN connectivity protocols, one dedicated WAN port and 2 LAN ports also configurable as LAN with claimed load-balancing and according to the vendor, "Abundant Security Features" [6]. We also found it interesting that it supports authentication via Facebook, or according to the firmware also WeChat.

The device contains a MIPS revision 1 based dual-core SoC with integrated Gigabit switch, 256 MB RAM and 128 MB flash capacity. The internal PCB shows signs of a debug UART serial port (see Figure 3.3).

Firmware analysis

Listing 3.3:]Binwalk output on router firmware image, omissions denoted by [...]

```
DECIMAL       HEXADECIMAL     DESCRIPTION
--------------------------------------------------------------------------------
70115         0x111E3         uImage header, header size: 64 bytes, [...]
    image type: Firmware Image, compression type: none,
    image name: "U-Boot 2018.09 for mt7621_nand_r"
404127        0x62A9F         CRC32 polynomial table, little endian
405151        0x62E9F         CRC32 polynomial table, little endian
470915        0x72F83         Flattened device tree, size: 3618 bytes, [...]
594403        0x911E3         Flattened device tree, size: 1853072 bytes, [...]
594631        0x912C7         LZMA compressed data, properties: 0x6D, [...]
2435723       0x252A8B        Flattened device tree, size: 10943 bytes, [...]
2494947       0x2611E3        Squashfs filesystem, little endian, [...]
20451924      0x1381254       Boot section Start 0x42424242 End 0x22313233
20451932      0x138125C       Boot section Start 0x0 End 0x0
```

The firmware of the device can be downloaded from the manufacturer's website. We inspect the contents of the binary file ('.bin' extension) using the open-source tool binwalk. The firmware contains an uImage-format U-Boot bootloader image, for flashing the actual bootloader and kernel image. Furthermore, we see LZMA compressed data (the kernel) and a Squashfs filesystem. SquashFS is an immutable compressed filesystem and can be extracted using binwalk or other tools. This filesystem

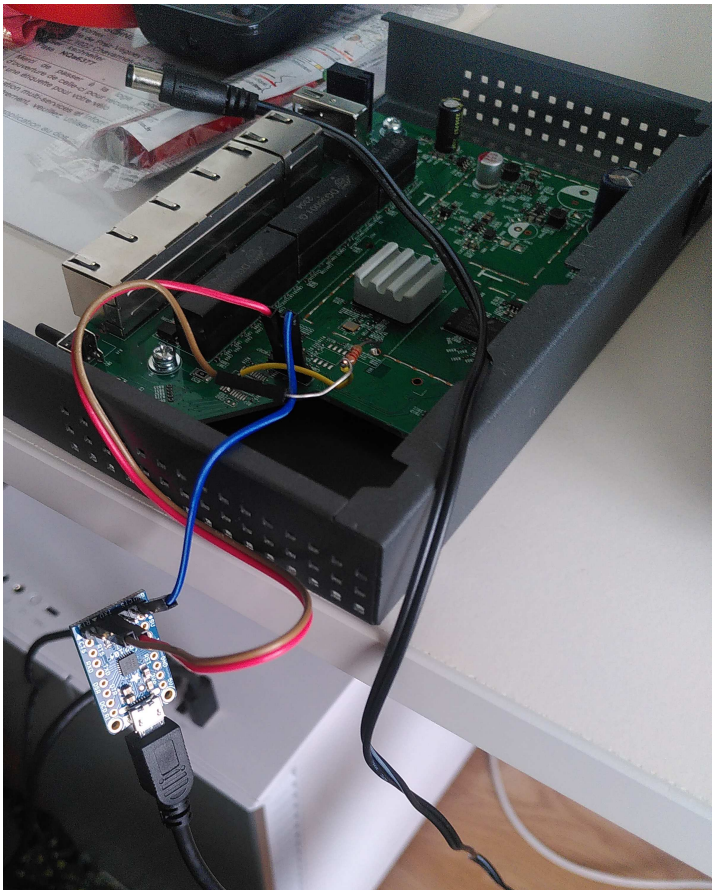

Figure 3.3: Accessing the serial port of the ER605 router

contains the root directory of the filesystem on the running router. To have persistent configuration, the router mounts a partition of the NAND flash and it uses OverlayFS as a union mount filesystem. Inside the filesystem, we can promptly learn that this device is using OpenWrt 14.07 as OS (from the file `etc/openwrt_release`). We continued exploring this firmware image and were able to find most custom parts of the firmware, firewall configuration and confirmed the baud rate of the serial port for later debugging (115200 baud, a common number that also the camera uses). Since OpenWrt is GPLv2-licensed, TP-Link is required to publish source code of the firmware running on the router. We can download an archive of the source code online, it seems to be slightly outdated but is still useful, since it contains build scripts, details on the firmware image format and custom packages, custom patches that differ from the official OpenWrt sources. Using the listing of patches applied to the system's packages, we were able to rule out known vulnerabilities of programs like the DHCP client udhcpc. Some custom parts of the firmware were missing from the published GPLv2 source code of the firmware, including the implementation of the web interface of the router in Lua. The firmware image contains Lua scripts compiled to Lua bytecode. This Lua bytecode is complicated to analyze, since Lua bytecode is platform-specific. Furthermore, OpenWrt uses a custom Lua compiler. We cloned Lua source code from the official repository of the version used on the router, applied patches from OpenWrt source code files and then used the compilation results to run luadec, which partially decompiled the custom Lua bytecode to Lua pseudo-source code. Since OpenWrt 14.07 was released in 2014, there are numerous such vulnerabilities of critical components of a router, that had to be patched and it is not a simple task to ensure that no issue was missed. Also due to this old version, our expectations for the security of this device were low: It runs a Linux 3.10 kernel, no ASLR / KASLR and barely any other protections, almost every process runs as root. Any code injection in a custom binary running on the device would result in achieving code execution with full priviledges. The strongest protection the Omada router had was a restrictive and basic default configuration. All Cloud functions were disabled and had to be enabled by user interaction, VPN clients were not running, no ports were listening on the WAN port (target of the competition) except for DHCP, DNS and HTTP ports, the latter being used for a connectivity check. Since we assumed, that it is required to successfully attack the device in the default configuration (no user interaction), although this later turned out not to be completely accurate, we quickly ran out of possible attack surfaces to test. The DHCP client was an old version, but all known vulnerabilities were patched and not in scope of the competition anyway. We attempted to fuzz the DHCP client binary (udhcpc, from busybox) but since it used raw sockets, there was no straightforward way to attach it to a fuzzer like AFL++ using e.g. hooking socket() calls with a `LD_PRELOAD`-ed library. Unfortunately, due to time constraints, we could not explore this path further. However, a promising future approach we envisioned was a custom compilation of busybox's udhcpc with coverage information and patches to allow reading DHCP packets from stdin, making it suitable for fuzzing.

```
local_3c.tv_usec = uac_com_get_dev_mac(&mac);
if (local_3c.tv_usec != 0) {
  if (cli_err_switch == 0) {
    return 0;
  }
  printf("[CLI_ERROR]%s()/%d read mac addr failed\n","cmd_debug",0x33c);
  return 0;
}
snprintf((char *)hash_input,0x17,"%02X:%02X:%02X:%02X:%02X:%02Xadmin",mac & 0xff,
         mac >> 8 & 0xff,mac >> 0x10 & 0xff,mac >> 0x18,(uint)(byte)local_30,
         (uint)local_30._1_1_);
len = strlen((char *)hash_input);
md5hashfunc(hash_input,len,pwd_out,0x20);
```

Figure 3.4: Decompiled password generation routine

Serial port shell

The UART serial port on the internal PCB of the router exposes the same shell as the SSH remote access (disabled by default from WAN) on the router does. The hardware for the UART port was incomplete. The correct pins were be identified using a Multimeter, to test voltage levels, and comparing present voltages to known values for UART. While it was possible to solder ground, Serial receive (RX) and Serial transmit (TX) pins to the pads, it was only possible to receive data, including the U-boot log and the Linux boot log, it was not possible to send data to the serial port out of the box. The culprit was eventually found on the PCB itself, a set of pull-up resistors and solder bridges, i.e. connection of two wires, between the CPU and the solder pads was unpopulated. We manually soldered in a pull-up resistor for the pin that was missing it and were now able to interact with the routers serial shell. Furthermore, the serial port was useful for interrupting the U-boot process and viewing the Linux boot log at startup. The serial shell was restricted to pre-programmed commands that could only change network settings etc. To access a fully-capable root shell, entering 'debug'-mode on the serial shell and entering a password was required.

The decompiled code in Figure 3.4 generates the password for accessing the root shell. The function `uac_com_get_dev_mac` is OpenWrt functionality to read the device MAC address. The MAC address is written into a string in the format "XX:XX:XX:XX:XX:XXadmin", and this is hashed by a function we named `md5hashfunc` that uses standard MD5 to generate a hash. This hash written in hexadecimal format in lowercase ASCII ("xxxxxxxxxxxx") is the password required to enter the unrestricted root shell on the device. Presumably other participants of the competitions also found this, since there now exists a web-page online [5] to generate this password for ER605 routers to easily root them via the normally restricted SSH management shell. This way of accessing a root shell on the device was much faster than our earlier methods, which involved interrupting the boot process, loading a custom U-boot environment to the device using 'kermit' tool and the serial port, then loading this environment to modify the kernel command line and boot it.

Chapter 4

Issues found in the Wyze Cam v3

Our research on the IoT camera concludes that the Wyze Cam v3 does not currently run secure software that is very open to exploitation once an initial attack vector is found. Concretely, we found a bug of questionable exploitability in the Realtek wireless driver that the Wyze Cam v3 ships with. This bug allows us to remotely force a crash and reboot of the device. Repeating the attack results in Denial of Service. We found a Shell Injection vulnerability in the setup procedure of the device, triggered by pressing an external Setup button and scanning a malicous QR code with the camera. In scenarios where the camera is deployed on the outside of some physically secured area, e.g. on the outside of a house, an attacker could also swap out the SD card and boot a backdoored Linux kernel using the `factory_t31_ZMC6tiIDQN` file. The Shell Injection vulnerability is located in the `libwyzeUtilsPlatform.so` library used by the 'iCamera' binary running on the camera. The intended Setup procedure behaviour is as follows: The user presses the Setup button on the outer case of the camera and is prompted to scan a QR code generated by the Wyze mobile app. We reverse-engineered the contents of these QR codes to be able to generate our own. It contains the Wi-Fi SSID and WPA password to the AP the camera tries to connect to, both are Base64 encoded. The password is additionally XORed with a string starting with "Wfb86GZX82JbwzW[...]", we believe this is an example of attempted security by obscurity. The QR code also contains information on the timezone and region. The injection is via the Wi-Fi SSID:

```
snprintf((char *)&command, 1019,
  "iwlist␣wlan0␣scan␣|␣grep␣\'ESSID:\"%s\"\'",
  &DAT_0001747c);
r = exec_shell_sync(&command, &outp);
```

The Wi-Fi SSID read from the QR code is used directly in a shell command. The function called `exec_shell_sync` runs the command specified in the first argument in the default shell `/bin/sh`. In an SSID chosen by the attacker, it is possible to terminate the string inside the command using `"` and `'`, then run arbitrary shell commands.

The bug found in the Realtek driver for the RTL8189ftv lies in the functions `rtw_aes_decrypt` and `aes_decipher`. According to our version of the driver, the function `aes_decipher` does AES CTR mode in-place decryption on a payload of length `plen` and calculates internally the number of blocks as `(plen-8) / 16`. This calculation is only correct, if we assume that `plen` is at least 8. The integer underflow of the subtraction causes the bug: The callee function `rtw_aes_decrypt` hands the exact 802.11 frame payload and payload length to `aes_decipher`, even if the payload is less than 8 bytes, i.e. a packet marked as 'Protected' does not contain 8 bytes of CCMP parameters. The number of blocks (`num_blocks`) is then used in the main decryption for-loop:

```
for (i=0; i< num_blocks; i++)
{
    construct_ctr_preload(...);
    aes128k128d(key, ctr_preload, aes_out);
    bitwise_xor(aes_out, &pframe[payload_index], chain_buffer);
    for (j=0; j<16;j++) pframe[payload_index++] = chain_buffer[j];
}
```

For a packet with a payload of length of 7, this results in an Out-of-bounds write on `pframe`. Since in C integer arithmetic, $7-8 = 0\texttt{xffffffff}$ the function will attempt to decrypt the entire memory above pframe in 16-byte blocks. On the camera, this quickly results in a crash:

```
[  384.753713] CPU 0 Unable to handle kernel paging request at virtual address
    00000004, epc == 8023dcfc, ra == 80073840
[  384.764712] Oops[#1]:
[  384.767067] CPU: 0 PID: 741 Comm: ksdioirqd/mmc1 Tainted: G [...]
[  384.776433] task: 80d74d80 ti: 85118000 task.ti: 85118000
[  384.782017] $ 0   : 00000000 10001c00 8a67df01 00000000
[...]
[  384.819951] $28   : 85118000 851197d0 851197d0 80073840
[  384.825371] Hi    : 00000000
[  384.828346] Lo    : 00000000
[  384.831334] epc   : 8023dcfc rb_insert_color+0x3c/0x154
[  384.836740]     Tainted: G           O
[  384.840624] ra    : 80073840 __enqueue_entity+0x84/0x98
[  384.846029] Status: 10001c02 KERNEL EXL
[  384.850090] Cause : 40808008
[  384.853065] BadVA : 00000004
[  384.856043] PrId  : 00d00100 (Ingenic Xburst)
[  384.860547] Modules linked in: [...]
[  384.880400] Process ksdioirqd/mmc1 (pid: 741, [...]
[  384.889401] Stack : [...]
          ...
```

```
[   384.926265] Call Trace:
[   384.928794] [<8023dcfc>] rb_insert_color+0x3c/0x154
[...]
[   385.014557] [<8001d640>] ret_from_irq+0x0/0x4
[   385.019379] [<c09467e4>] mix_column+0x98/0x170 [8189fs]
[   385.025404] [<c09469f0>] aes128k128d+0x134/0x1ac [8189fs]
[   385.031616] [<c0947cdc>] aes_decipher+0x21c/0x934 [8189fs]
[   385.037918] [<c0949900>] rtw_aes_decrypt+0x274/0x374 [8189fs]
[   385.044498] [<c097e5b0>] decryptor+0xd8/0x158 [8189fs]
[   385.050460] [<c0981e20>] recv_func_posthandle+0x28/0x100 [8189fs]
[   385.057401] [<c0982098>] recv_func+0x1a0/0x1cc [8189fs]
[...]
[   385.106624] [<8006324c>] kthread+0xbc/0xc4
[   385.110866] [<8001d6fc>] ret_from_kernel_thread+0x14/0x1c
[   385.116450]
[   385.117986]
Code: 30660001 54c00044 03c0e821 <8c660004> 50460017 8c660008 50c00007 8c460004
8cc80000
[   385.128306] ---[ end trace aefc2ed88241df05 ]---
```

Creating an exploit for this vulnerability is challenging for multiple reasons. Since the packet triggering the crash can only contain less than 8 bytes, the attacker would need to place shellcode in memory of the device beforehand. Furthermore, the bug stores the result of AES CTR mode decryption in Out-of-bounds memory, meaning existing data is XORed with some possible predictable, but constantly changing value. To reliably place desired values in specific memory location would require a very precise and large memory content leak vulnerability. In the limited timeframe of this project, it was not possible to develop such a sophisticated exploit required to use this bug to gain arbitrary code execution. Nevertheless, this is a serious Denial-of-service vulnerability, present in many related versions of this Realtek wireless driver.

Chapter 5

Conclusion

In our analysis of the security of the Wyze Cam v3 for the Pwn2Own 2023 Toronto competition, we found two vulnerabilities, in custom Wyze software and kernel modules used on the device. They can be exploited to gain remote code execution and to cause a software crash of the Operating System of the camera. We also analyzed the TP-Link Omada ER605 router, gaining insights on hidden debugging capabilities. The Wyze Cam v3 has a significantly reduced attack surface by consistently using HTTPS and DTLS encrypted and authenticated communication channels. Others showed that this authentication can be bypassed and is not enough to protect against attacks [13]. The bug we found in the Realtek Wi-Fi driver demonstrates that relying on insecure closed-source device drivers is risky, especially for IoT devices. Our analysis further revealed that the Wyze Cam v3 software features very little protections against attacks and further undiscovered security issues likely exist. To improve security and minimize risk, Wyze and IoT vendors in general need to focus on implementing modern security features, writing secure custom software and keeping device drivers secure and up to date, rather than adding new features and relying on cryptography to protect against exploitation.

Bibliography

[1] Federal Trade Commission (Consumer Alert). *Ring's privacy failures led to spying and harassment through home security cameras*. https://web.archive.org/web/20230929013057/https://consumer.ftc.gov/consumer-alerts/2023/05/rings-privacy-failures-led-spying-and-harassment-through-home-security-cameras. Accessed: 04.01.2023.

[2] Bitdefender. *Smart TVs Rely on Vulnerable Technology, Insecure Code*. https://web.archive.org/web/20220820232659/https://www.bitdefender.com/blog/hotforsecurity/smart-tvs-rely-vulnerable-technology-insecure-code/. Accessed: 04.01.2023.

[3] Bitdefender. *White paper: Vulnerabilities Identified in Wyze Cam IoT Device*. Tech. rep. Mar. 2022. URL: https://www.bitdefender.com/files/News/CaseStudies/study/413/Bitdefender-PR-Whitepaper-WCam-creat5991-en-EN.pdf.

[4] Marc Egli. "Over-the-Air LTE Protocol Fuzzing". MA thesis. Daejeon, Republic of Korea: EPFL, KAIST, Aug. 2023.

[5] *ER605 v2 Root Password Generator*. https://chill1penguin.github.io/er605v2_openwrt_install/er605rootpw.html. Accessed: 04.01.2023.

[6] TP-Link. *ER605 Omada Gigabit VPN Router*. https://www.tp-link.com/us/business-networking/omada-sdn-router/er605/. Accessed: 04.01.2023.

[7] Forbes Magazine. *Coffee Machine Hit By Ransomware Attack—Yes, You Read That Right*. https://web.archive.org/web/20221129072901/https://www.forbes.com/sites/daveywinder/2020/09/27/hacker-takes-coffee-machine-hostage-in-surreal-ransomware-attack/?sh=7aba1e4077f0. Accessed: 04.01.2023.

[8] Tony Nasr, Sadegh Torabi, Elias Bou-Harb, Claude Fachkha, and Chadi Assi. "Power jacking your station: In-depth security analysis of electric vehicle charging station management systems". In: *Computers & Security* 112 (2022), p. 102511.

[9] Securityweek. *TP-Link Smart Bulb Vulnerabilities Expose Households to Hacker Attacks*. https://web.archive.org/web/20231011060716/https://www.securityweek.com/tp-link-smart-bulb-vulnerabilities-expose-households-to-hacker-attacks/. Accessed: 04.01.2023.

[10] Vijay Sivaraman, Hassan Habibi Gharakheili, Clinton Fernandes, Narelle Clark, and Tanya Karliychuk. “Smart IoT devices in the home: Security and privacy implications”. In: *IEEE Technology and Society Magazine* 37.2 (2018), pp. 71–79.

[11] *static-binaries GitHub repository*. `https://github.com/andrew-d/static-binaries`. Accessed: 04.01.2023.

[12] Techcrunch. *Wyze launches its new $34 pan and tilt security camera*. `https://web.archive.org/web/20230302235922/https://techcrunch.com/2023/01/10/wyze-launches-its-new-34-pan-and-tilt-security-camera/`. Accessed: 04.01.2023.

[13] *unwyze - a Wyze Cam v3 RCE Exploit*. `https://github.com/blasty/unwyze`. Accessed: 04.01.2023.

[14] Wyze. *Response to 3/29/22 Security Report*. `https://web.archive.org/web/20231113140201/https://www.wyze.com/pages/response-to-3-29-22-security-report`. Accessed: 04.01.2023.

Appendix A

List of acronyms

IoT Internet of Things 4

DDoS Distributed Denial of Service 4

SoC System-on-Chip 5

OS Operating System 5

RAM Random-access memory 5

SOHO Small Office / Home Office 6

RISC Restricted Instruction Set Computer 7

RX Serial receive 21

TX Serial transmit 21

UART Universal asynchronous receiver-transmitter 9

SDN Software-defined Networking 18

OEM Original equipment manufacturer 8

PCB Printed circuit board 9

GPIO General-purpose IO 13