Investigating todays IoT Security through an IP camera

– or –

Adding some 60k devices to your bot net

Marvin Bornstein Balthasar Martin
{marvin.bornstein, balthasar.martin}@student.hpi.de

Hasso-Plattner-Institute

1 Introduction to the Internet of Things

Humans have been using lots of devices and gadgets to ease and improve their daily life. This includes light and heating to feel comfortable, kitchen equipment for cooking, cars, trains and planes for transportation, medical devices, like pacemaker, for regaining the ability of living as well as computers and smart phones as access to information and entertainment.

1.1 IoT as a novel paradigm

Because of the increasing amount of devices and overhead of controlling them, the idea of smart environments arose. Devices should adapt to how they are used and remove tedious control from the user to an intelligent managing entity or the device itself. The Internet of things (IoT) is a novel paradigm that is hoped to pave the way for smart environments by integrating several technologies and communication schemes [1]. The fundamental idea is to equip each device with radio communication, tags and sensors, make them uniquely identifiable and interconnect them with their environment, like neighboring devices. Making them interact with each other and cooperate to reach common goals.

> IoT is included by the US National Intelligence Council in the list of six “Disruptive Civil Technologies” with potential impacts on US national power [2]. NIC foresees that “by 2025 Internet nodes may reside in everyday things – food packages, furniture, paper documents, and more”. [3]

1.2 Properties and challenges of IoT

Most of the ideas around IoT involve lots of sensors and actuators. They could be measuring energy consumption or water plants. For many tasks, they are required to be really small and keep working without power supply for a while. So the devices need to work with very restricted resources and this is what most people immediately think of IoT. However, there is much more to it. Think of the free market and all the different parties that want to sell and manufacture IoT devices. The landscape of products will vary hugely. Vasilomanolakis et al. [4] present the following 4 challenges:

Uncontrolled environment Devices can be mobile, i.e. they move around. This means that they can loose their network connection and they cannot be supplied with power by cable. As they could be installed in public space, attackers may physically access or damage devices.

Heterogeneity There are lots of areas where IoT can be employed and by digitization of current processes and companies, a huge variety of tasks need to be addressed. Eventually, this leads to a multitude of "things" that can all come from different manufacturers. Interoperability and compatibility gains importance.

Scalability The huge amount of nodes with high interconnectedness can be too much for centralized infrastructures. Scalability will be so important, that current approaches need to change or cannot be used anymore.

Constrained resources Restrictions in size and energy consumption will limit the available hardware resources. This is not only a step back in terms of computational power, but also an significant aspect for radio hardware and protocols. Battery technology is limited and anti proportional to physical size.

1.3 IoT design ideas

To account for all the requirements, there are some design ideas and technologies – e.g. adding additional layers of hierarchy. Sensors and actuators may be controlled by a *managing node* and they communicate on low energy hardware with special protocols. Another idea is to design multi-hop networks. The managing device, however, can implement more demanding standards and protocols, such as the current TCP/IP stack. Standards need to make sure to implement an *abstraction layer* between all the different devices and the user's task should be to compose IoT services. Today, there are only few accepted standards developed and implemented – most of the vendors have their own protocols and requirements. In combination with economic pressure to release new products as fast as possible, there may be increased potential for security vulnerabilities.

2 Related work

Network cameras as embedded devices that are connected to the Internet in high numbers and have a potential privacy impact are interesting hacking targets – for researchers and fraudsters alike. Past attacks on such devices include concepts as simple as scanning the Internet for exposed camera web interfaces and brute forcing access with default credentials. Results are presented on various web pages [1] that collect open or badly secured camera interfaces so they can be viewed by the visitors of said pages. Thereby the attack vector of weak passwords is the same as for Mirai with the difference just lying in HTTP instead of telnet.

Many security researchers have also looked at IP cameras. They mostly focus on exposed web and RTSP interfaces and are not limited on the cheapest vendors (e.g. the Hikvision research of Core Security)[5]. Such interfaces can be Internet exposed, unknowingly to the user, thanks to UPnP functionalities and automatic port forwarding [6].

Most recently, Pierre Kim discovered an authentication bypass in the GoAhead web interface that is or was used in network cameras of the majority of vendors [7]. Combined with an already known command injection this could be used for remote exploitation.

This command injection was found by Zoltan Balazs and enabled finding a telnet backdoor account. The same research also includes first looks into a cloud based UDP protocol.[8]

As past research mostly focused on Internet exposed web interfaces and our device (running the latest firmware version) only shows left over traces of a HTTP server and does not offer a ftp functionality, we focused our research on new ways of exploitation and the UDP cloud protocol.

Also, the premium vendor Hikvision is discontinuing his dynamic DNS service while transferring to a non-IP-based solution [2], so we believe these are the functionalities the whole industry is heading to.

3 Security concerns and implications

The Internet of things with it's novelty and special properties (see Section 1.1) introduces new security challenges while having a big potential impact if compromised.

3.1 Security challenges in the Internet of things

The need for low energy consumption and limited computation power can eliminate the possibility to use asymmetric cryptography [9]. This can prevent the use of existing trust structures like the public key based ecosystem of TLS certificates and certification authorities.

[1] E.g. https://www.insecam.org/
[2] http://www.hikvision.com/en/faq_81.html

Symmetric keys that are on the device may be extracted which can allow man in the middle attacks or the creation of malicious firmware files. An example for this are Ronen et al. who extracted the AES firmware key out of a Phillips Hue smart lamp using sophisticated power analysis [10].

Providing a firmware update mechanism is vital to patch potential security vulnerabilities, because most devices are still in active development and not extensively tested. With the potential high number of devices, this has to scale well [4] while it should not need any advanced user interaction as the devices are used by non-technical users.

Also, as smart devices may be employed in public space, the attackers ability to physically access his target devices may need to be considered. [4]

A broad selection of different devices, manufactures and standards makes it hard to keep track of security vulnerabilities and patch statuses.

3.2 Impact of compromise

In the future, IoT devices may be integrated in most parts of our life, be it door locks, fridges, home control systems, smart energy devices or traffic control sensors. For the end user this may enable privacy violations, because an attacker could gain information and potentially control about every aspect of his victims life.

For state-level actors this could introduce new ways to monitor, spy on, manipulate and oppress own or foreign citizens.

The sheer number of IoT devices makes them interesting targets for fraudsters. The compromise of one device type can enable the creation of huge bot nets that may be used for commercial DDoS attacks like they are done on the basis of Mirai.

Mirai powered attacks against the web site of journalist Brian Krebs [3] with a bandwidth of more than 620 Gbps introduced the content delivery network Akamai to a new size of DDoS attacks. Although their biggest attack experienced to this date was close to half the size with 363 Gbps, the Mirai attacks were – in contrast previous attacks – not amplified via DNS or NTP [11]. The same bot net code was also used in attacks against the DNS provider DynDNS which resulted in many major web pages (i.e. their domain name records) being inaccessible in big parts of the United States and other parts of the world. [12]

Other destructive consequences of hacks are demonstrated by the Phillips Hue research of Ronen et al.: because of their mesh network communication, the infection of one smart lamp can be escalated to smart lamps of a whole city if their density is high enough. By simultaneously turning them off and on again an attacker could stress the city's power grid with a power outage as a potential consequence[10]. With IoT devices being employed in infrastructure like traffic systems and the power grid, one can imagine incredibly damaging attacks, e.g. in combination with black mailing attempts.

[3] https://krebsonsecurity.com

4 IoT Security case study - audit of a network camera

4.1 Motivation for choosing the camera

The *Mirai bot net* showed, how many insecure devices there are. We could only focus on one device and IP cameras have made up most of the bot nets devices. Also, IP cameras often come with Linux, quite good hardware and work on common TCP/IP protocols. Although this is not the vision of IoT, it is very common, because it allows smartness without too much new development. Also, smart home devices like lamps usually rely on managing devices and have a more complex setup. And since there are not many global standards or vendor still implement their own, analysis results may not apply to other vendors or products. Thus, network cameras are quite interesting targets at the moment. With unauthenticated telnet access being an easily patchable vulnerability, we wanted to find out whether other fundamental flaws in the design and implementation of IP cameras exist.

When looking at available camera manufacturers and models, a lot of them seem similar (see Fig. 1). The same applies for the management apps (see Fig. 2). This means, they are either from the same company or subsidiaries to have a higher market share or form other companies, but share components and implement the same standards. We assume, that some of the security vulnerabilities apply to several of these devices.

4.2 Feature overview and setup of the Sricam

In the eyes of a user, the cameras offer similar features. They include a camera, a microphone and a speaker. It can record audio and video that one can watch on the smart phone from everywhere over the Internet. It also has an infrared sensor and infrared LEDs to enable night vision. It can detect motion and notify the user by sending an email. The user can record their voice and let the camera play it.

The initial setup and all other settings must be set using the app. In order to use the app, the user needs an account and an email address. After the login, the user can add devices, i.e. cameras, to the account. The camera needs to be powered on and connected to the network by Ethernet. It has a default password, which the user is recommended to change. However, changing the default password can also be skipped. After the device is added to the account and visible in the app, the user can select the WiFi network it should use. The camera also offers the following settings:

- changing the time
- setting the video format and recording volume
- adding visitor passwords / changing the main password
- IP and DNS settings
- motion alarms: email, buzzer alarm, motion detection settings
- recordings to a SD-card

Fig. 1. A subset of network IP cameras on eBay[5]. Looking alike, having similar features and product designs

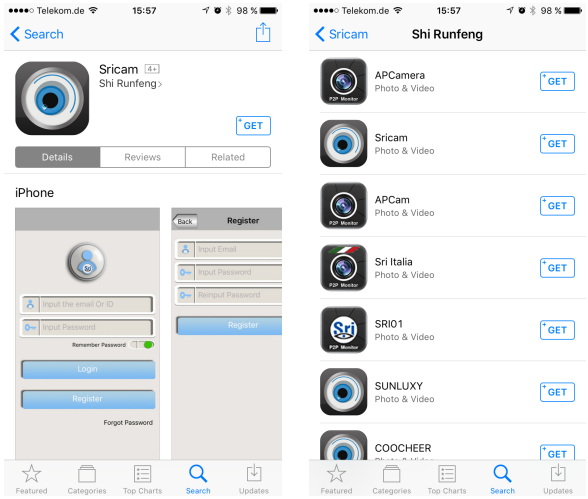

Fig. 2. Apps in the Apple App Store and apps by the same developer of the Sricam app

– labeling the recorded area
– storage settings
– device update

4.3 Implementation analysis and device reconnaissance

As for the hardware, the camera features an *Ethernet* port, a *microSD* slot and DC power interface. Dismantling the camera reveals a *Grain GM8135S*[6] SoC with a 32-bit ARM CPU and 512 MB DDR RAM (see Fig. 3). It also has a serial interface, that we connected to using a serial to USB adapter and *picocom*[7] software (see Fig. 4).

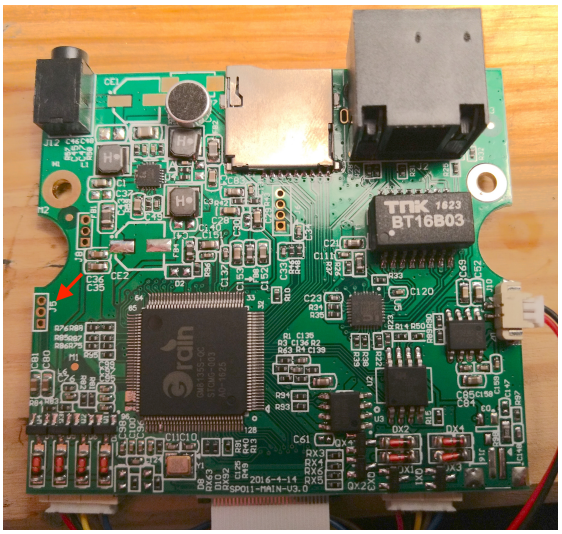

Fig. 3. System on chip of the camera. The three ports marked with an arrow provide the serial interface.

Software-wise, the camera runs a *BusyBox* Linux and includes telnet, which usually does not run. However, via the serial shell we added the execution of the telnet daemon to an initialization script in order to have easy shell access. Almost all of the camera logic is implemented in a 32-bit executable, called *npc*. The camera uses HTTP to check for new firmware versions and downloads them via HTTP. Streaming the live camera video is based on *RTSP*. All other control commands from the app are sent over UDP. When the app and the camera are communicating outside the same local network, the app sends the packets to an API server who relays them to the camera. This will even work if the camera

[6] `http://www.grain-media.com/html/8136S_8135S.htm`
[7] `https://github.com/npat-efault/picocom`

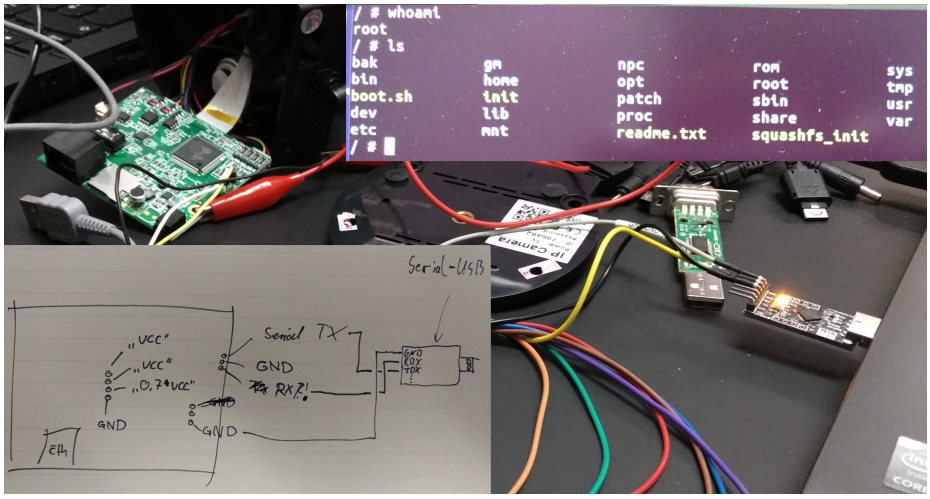

Fig. 4. Connecting to the boards serial interface. Login as root without credentials.

is behind a router with a firewall. We will describe the communication in more detail later, the camera basically keeps the UDP connection and NAT translation open by regularly contacting the API servers. The app uses UDP for the camera communication and HTTP to authenticate against the API servers. Most of the information, such as all devices and WiFi passwords are sent to and stored on the API server for the user account.

4.4 Attack surface and audit concept

We want to test the security of the camera from different angles. An attacker usually has one of two goals in mind: stealing sensitive information or execute arbitrary code on the remote system. There are other goals in information security, but these two are the most severe. Additionally, there are different conditions for an attacker. The two that we focused on, are that one has control over the network or that one has not. By *control over the network* we mean being a node in between the communicating partners (i.e. *MITM*). While this is an easy position for exploitation and manipulation, it also is not the standard case. However, there are ways for an attacker to get into the position, so any system security needs to also cover this.

The attack surface matrix in Table 1 reflects these cases. We checked each box step by step and, consequently, we will structure the next sections accordingly.

	Information leaks, privacy violation	remote code execution
No network control	– Authorization against Sricam servers – Authorization against camera	– Exploit camera functionalities (e.g. buffer overflow in send mail alarm function)
Network control	– Sniffing camera feed / credentials – Sniffing non-camera secrets (WiFi passwords, ...)	– malicious software update

Table 1. Attack scenarios for different prerequisites and goals.

4.5 Information leak and low encryption

Starting with the lower left box of the matrix in Table 1, we investigated what kind of information are sent. Using the setup shown in Fig. 5, we found out, that the app-server-communication is mostly HTTP (for login, version checks; the same for camera updates). The communication between app and camera is mostly UDP and follows a proprietary protocol. When the server acts as a relay server, the app and the server also talk UDP. In general, most of the messages are unencrypted. This includes the camera feed, control commands and even the login credentials.

During setup of the camera and when configuring the WiFi, the camera sends the WiFi password to all API servers in plain text. Fig. 6 shows all of the involved servers, their IP and their location.

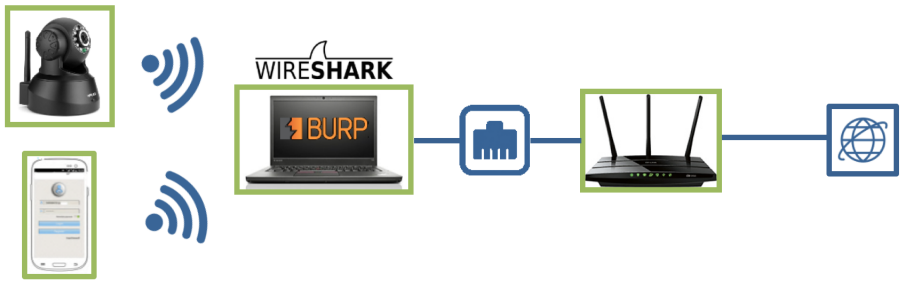

Fig. 5. Laptop acts as AP for both the smart phone app and the camera. It runs *Wireshark* and *BurpSuite* to see and alter TCP/IP packets.

Host name	IP address	location
{api1, p2p1}.videoipcamera.cn	101.1.17.22	Hongkong
{api2, p2p2, upg, upg1}.videoipcamera.{com, cn}	218.30.35.92	Shenzen
{api3, p2p3}.videoipcamera.cn	220.231.142.137	Shenzen
{api4, p2p4}.videoipcamera.com	146.0.227.241	Romania

Fig. 6. All of the companies servers and which domains point to them.

Section 4.2 mentions the motion detection email alarm feature. In order for it to work, the user has to provide an email account, i.e. the SMTP server IP address as well as the login credentials. The camera also sends this data to the API server. This time, however, it encrypts the password using DES and, as Fig. 7 shows, with a static key that can be extracted from the Android app.

```
public class DES {
    static byte[] key = new byte[]{(byte) -100, (byte) -82, (byte) 106, (byte) 90, (byte) -31, (byte) -4, (byte) -80, (byte) -126};

    public static byte[] des(byte[] str, int type) throws Exception {
        if (type == 0) {
            return desEncrypt(str, key);
        }
        return desDecrypt(str, key);
    }
```

Fig. 7. Decompiled Java app code (using jadx-gui) showing a hard-coded key.

4.6 Deploy malicious firmware

Information leaks and privacy violations are nice, but we really want to get into the camera. To do that, we try to get our own firmware onto the device by altering the update process. In general, this includes three steps:

1. get the camera's firmware
2. dissect the firmware, what it does and how it works
3. change the firmware so that it is still valid, but contains a backdoor
4. alter the update process to let the camera install the changed firmware

Getting the firmware After the user tells the camera to check for updates, it asks the update server for the latest version. If it is newer than the current one, it downloads and installs it. By using Burp Suite to alter the answer to the current firmware request, we get the device to try and download the new firmware file. Now we just need to go and download the current version file ourselves, the link is `http://upg.videoipcamera.cn/upg/14/00/npcupg_14.00.00.52.bin`.

The structure of the firmware Fig. 8 shows the general structure of the firmware file. It has three parts: a header, a file system and a binary. The header contains the sizes of the other parts of the file, a checksum and a version number. The JFFS2 file system contains all necessary files and the main executable, called *npc*, it gets mounted on the camera and replaces the old files. The executable binary is used as part of the update process.

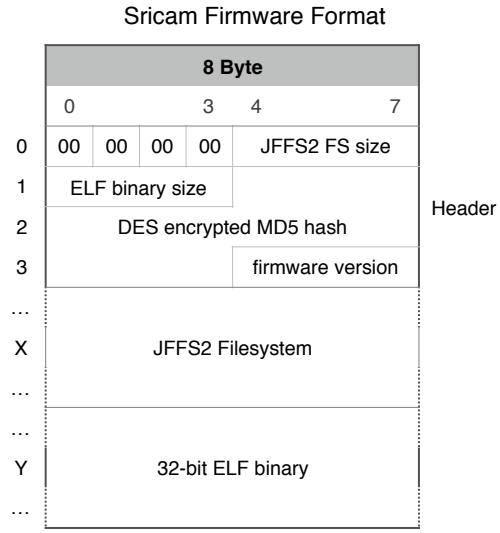

Fig. 8. Format of the firmware file. It consists of a header, a file system dump and an executable.

Altering the firmware As a proof of concept, we edited a boot script so that it starts the telnet daemon (see Fig. 9). We did this directly on the camera, so in order to create the JFFS2 dump, we only needed to *dd* from the mounted device to a file. Now we just stitch the JFFS2 file together with the 32-bit ELF binary and the header from the original firmware file. JFFS2 appends the most recent changes to the end of the file, so our changed file system dump is larger than the old one. Hence, we will have to update the size in the header. Also

part of the header is a checksum value. When we first created a firmware file, we did not know how it is computed and tried several checksum algorithms without successfully recreating the needed hash.

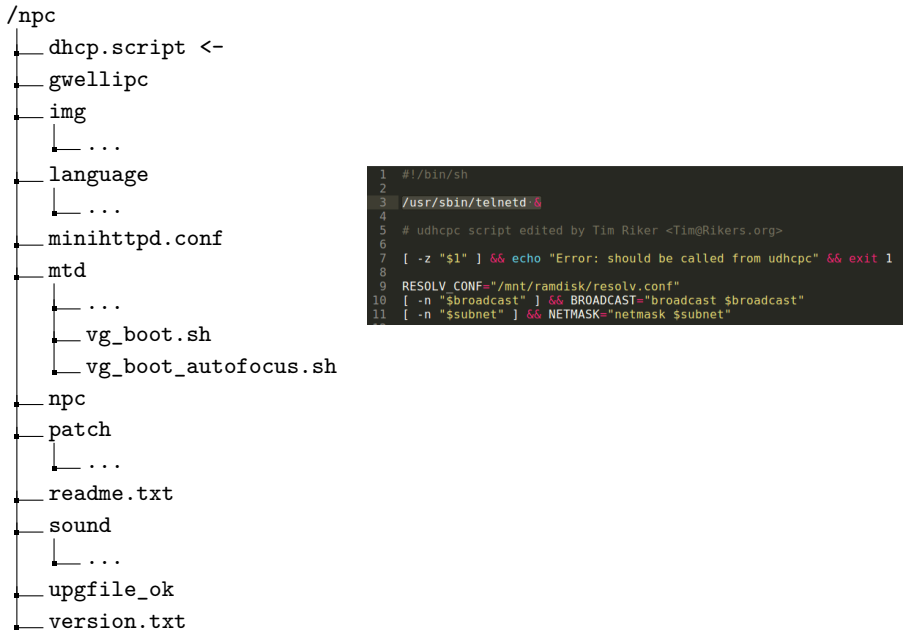

Fig. 9. Content of the file system left. On the right, the added line to create the telnet back door.

Getting the camera to download our firmware The firmware is not signed and the update process is built on plain HTTP. So we need to route the camera's update request to our web server. This could be achieved by adding an *iptables* rule on our access point (which is what we did in our local setup) or by changing the DNS settings of the camera (which is done in the demo in Section 5). A simple python script sets up a web server and answers update checks with newer versions and serves our firmware file. The "do firmware update" section in Fig. 13 shows this process.

Remaining debug output got us the checksum When the camera updates on a firmware file with an incorrect checksum, it prints (to its serial output) "Md5 err!". However, during the installation of an original firmware, the camera prints several numbers. These numbers turn out to be bytes in the firmware header. So we checked the disassembled code in the npc binary using *IDA* in order to understand how the checksum is calculated. The assembly code does

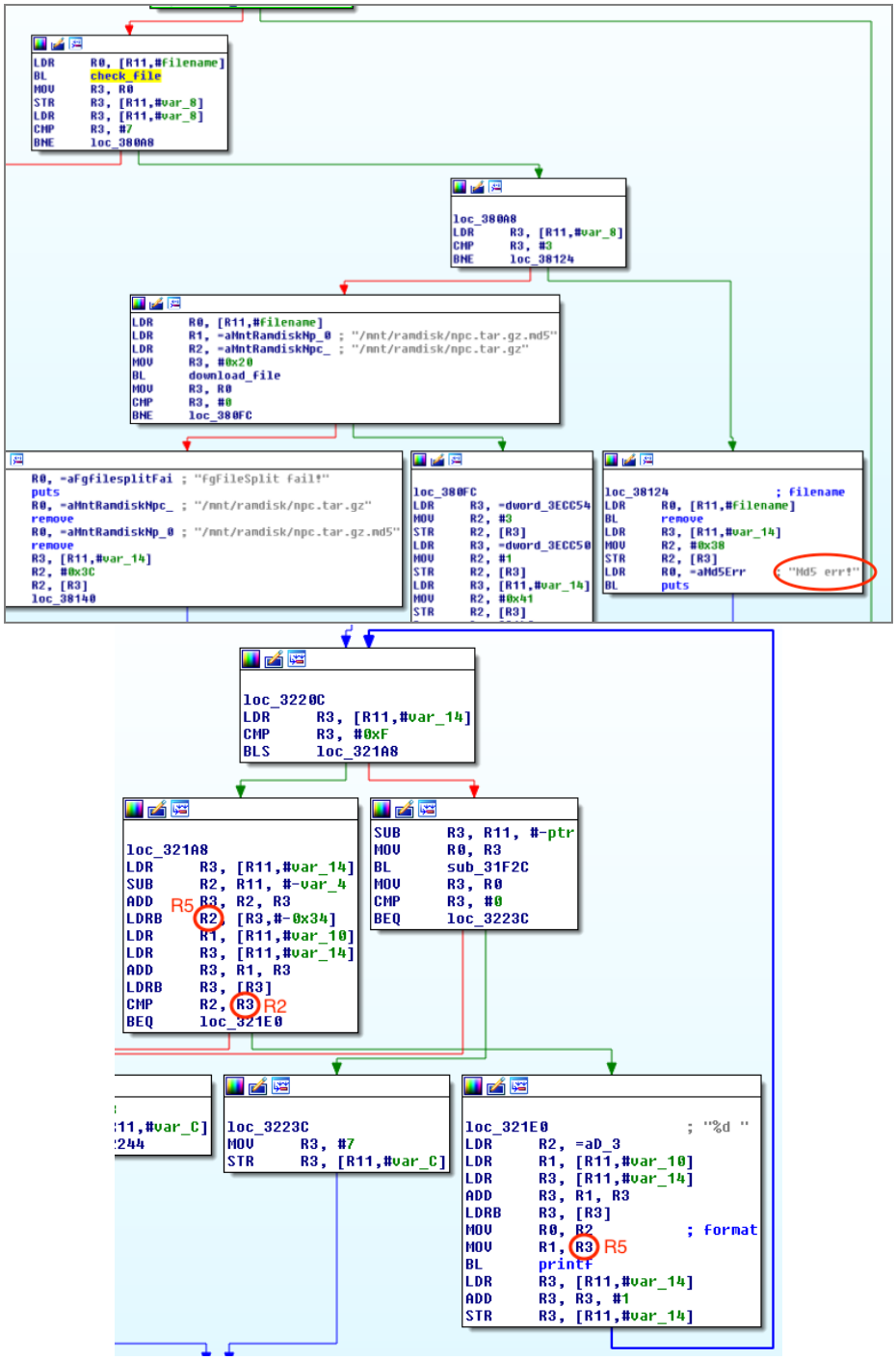

Fig. 10. Upper screen shot shows the code from checking the firmware file to the MD5 error. The lower one shows part that is marked as *check_file*. It checks the checksum byte by byte and prints each byte if it is correct. Changing the registers (to the one in red), makes it print the expected byte.

reveal some strings containing "MD5", but also does something else, that we could not figure out easily. Thus, we focused on the bytes that are printed – maybe it can tell us the correct checksum. We searched for the error message and the HTTP firmware download request strings. Fig. 10 shows the decompiled code of the npc binary. The code in the upper image calls a checking function and the lower image shows the checking function. It compares each byte of the checksum in the header with the expected byte, byte by byte. If the bytes match, the byte given in the header is printed to stdout.

So, as indicated in the lower image with the red circles, we patched code slightly to always print the expected byte (see Fig. 11). Finally, we ran the update with our malicious firmware and got the valid checksum calculated and printed to the serial output of the camera.

```
kill -9 [process_number]
printf '\x50' | dd bs=1 seek=172469 count=1 of=/npc/npc conv=notrunc
printf '\x02' | dd bs=1 seek=172488 of=/npc/npc count=1 conv=notrunc
printf '\x05' | dd bs=1 seek=172536 of=/npc/npc count=1 conv=notrunc
```

Fig. 11. Commands to patch the *npc* binary on the camera. The patches need to be done quickly after killing the process, because the system reboots.

Later, we found this Github repository for Gwell Media firmwares[8], that contains code to compute the checksum. It turns out to be a MD5 hash, DES encrypted in ECB mode. Had we known that the MD5 hash is encrypted using the exact same key as in Fig. 7, we could have saved some time and effort. Nevertheless, by patching the npc binary, we gained further insights in how the camera is working.

4.7 Authentication

The upper left box of the attack surface matrix in Table 1 deals with the Authentication mechanisms of the camera and the servers. We can observe two different kinds of authentication happening. When setting up his camera, the user has to add it to his sricam account specifying device ID and device password that are printed on a sticker on the device. The device ID is a six-digit integer and the default device password is 888888 for all devices.

On which ever smart phone he later logs into this sricam account, he will be able to access this device. That means we have to distinguish authenticating against Sricam servers to login and authentication that has to happen when control commands are send to the device.

[8] `https://github.com/zzerrg/gmfwtools`

Against Sricam login servers When logging into the account the app sends a HTTP POST request with the mail address and the MD5 hash of the users password to the login servers and if they are valid gets a session ID as response. It then requests the contact details for all devices associated with the account and that send to the server when the device was added. For each device this list consist of the device ID, the display name and the device password encrypted with the user ID of the account (see Fig. 12).

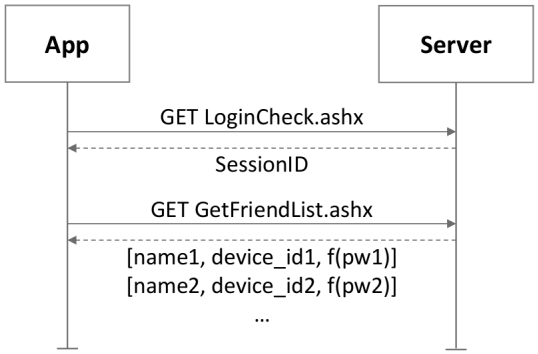

Fig. 12. The HTTP requests and responses when logging into a Sricam account to use the app.

A pre-compiled library is used for decryption within the app via `contact.contactPassword = P2PHandler.getInstance().HTTPDecrypt(str, contactPwd, 32)`. With this contact information the app is then able to control the device.

Beyond the login being plain HTTP and that it should easily be possible to run the decryption method with any potentially sniffed contact information, we did not find a vulnerability in the login functionality that would allow an attacker to access other accounts. Therefore we decided to take a look at how the app uses the contact information to control its linked devices.

In control pakets to the device The control of the camera – i.e. changing settings, moving the device, viewing the feed, ect... – happens via a proprietary UDP protocol. When the smart phone is in the same local network as the camera, both can communicate directly. If they are in different networks, the connection is relayed via the Sricam servers. As soon as the camera boots up, it starts sending UDP pakets to the servers and continues sending regularly to keep the entry in the NAT table of the router alive. This way the relay server can contact the device by just using the existing connection. No port forwarding is necessary at the router.

The pakets send in the local network contain a header that tells if they are a request or response and if they were send by the app or by the camera. Then

follows a command ID that falls into a different 1000 integer interval for each command (e.g. $50000 - 60000$ for `MESG_SET_IP_CONFIG`). The rest of the paket we call the payload and it contains data specific to the command. In the payload there may be a value for authentication included, but if so, it depends only on the device password and maybe the account ID, so it can be reused for different devices with the same password.

If the connection is relayed, 24 byte are added in front of the local paket. This includes the ID of the account sending or receiving the paket and the device ID. This structure is illustrated in Fig. 14. The app does not need to know the IP address of the camera, because the relay servers decide where to send the paket based on the included device ID. Also, the commands are just redundantly send to each of the for relay servers and the camera just answers the first one it gets to all relay servers as well.

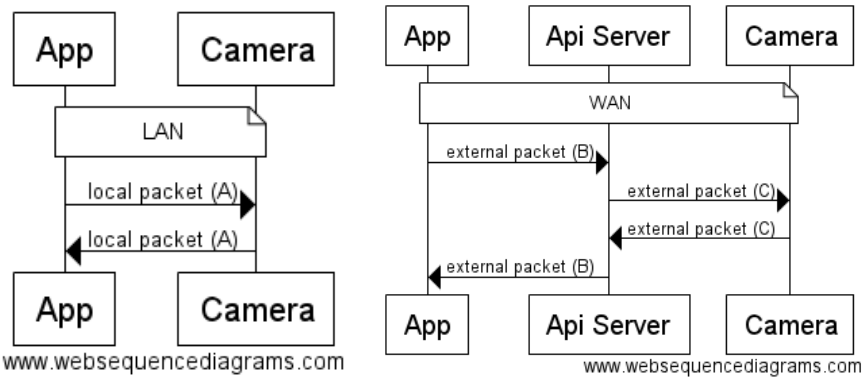

Fig. 13. Communication flow depending on the network setup.

We believe byte 12 to 23 to be some authentication value that depends on the device password and maybe the account ID. This value is modified by the relay server when he recieves a paket from the app or camera and forwards it to the other party (see Fig. 13). We are sure, that it does not depend on the device ID, because the same paket can be send to the relay server with a different device ID and will be relayed to the new device – given that the second device has the same password the paket will be accepted and it's action executed.

Given a target password, we can create control pakets for other devices by setting our device password accordingly, then executing the action on our device and sniffing the paket. After copying it into our python script `python_client.py` and replacing the bytes with our device ID with the target device ID, we can send the paket to the relay server. It will be forwarded to the target device that will execute the encoded action.

Most of this analysis was done via trial and error by capturing pakets in Wireshark, copying the UDP data as bytes into python and then modifying and resending them via a UDP socket.

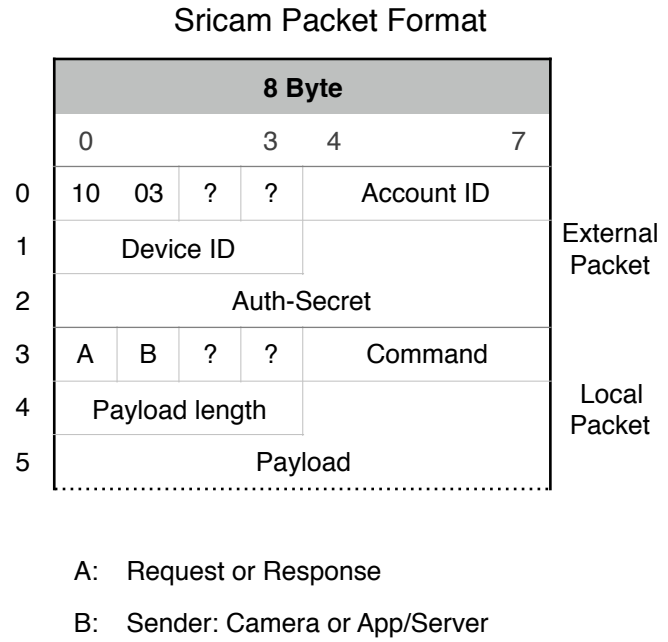

Fig. 14. Packet format of Sricam's proprietary communication protocol.

Device enumeration When a user first adds a device to his app, he is presented a window to change the password of the device. As this window contains a skip button, we suspected that a significant portion of users does not change the password. In contrast to owning all, for a bot net it is more important to own enough devices. After more or less understanding how control pakets are crafted, we therefore decided to try and enumerate all possible device IDs with the standard password.

In the main window of the app all devices associated with the account are listed, each with green, yellow or red dot next to it, indicating if the device is currently online. When refreshing this view, a paket containing all the device IDs is send to each relay server and the answers include per device if it is online. These pakets can contain up to 128 device IDs and can be replayed with other device IDs, independently from their passwords. As a online device does not necessarily appear online for every server their answers have to be aggregated. With our python script `find_devices.py`, we were able to enumerate all possible

device IDs from 0 to 999999 in one hour and found 140741 valid IDs of online devices.

Password enumeration The next step was to investigate which of these device use the default password. In the Java class with the command IDs we found a message called `MSG_ID_CHECK_DEVICE_PASSWORD`. A Wireshark filter for udp pakets that identified pakets which would have a command ID in the respective range helped us find out that such pakets get send as soon the settings for a camera are opened in the app. Changing the password of our device and replaying the original check password paket helped us identify which byte of the answer indicates if the password is correct. Checking if a password is correct for a specific device ID is now as easy as setting our device to the specified password, capturing such a paket and resending it to the relay server with a changed device ID. We wrote a threaded python script that checks all of the online device IDs for the default password and collects the results. While running this script, it was necessary to open an access point on the laptop and connecting the smart phone with the running app to it. Probably to make it look like the requests come from a legitimate app.

We found that 63029 devices are accessible via the default password.

4.8 Potential consequences

Thereby we could access the video feed of 63029 IP cameras of which a significant portion will be in private homes. We could also record their audio and send our own audio to their speakers.

Another possibility is harvesting SMTP credentials of the users who enabled the mail alarm feature in these cameras.

Last but not least, we can automate remotely installing our malicious firmware update on all of these devices like it is demonstrated in Section 5 and build a 63029 devices bot net. With the attack on DynDNS in 2016 – that rendered many major web sites unusable – being reported to have up to 100000 devices involved, such a bot net could cause incredible damage.

4.9 Summary, mitigations and conclusion

This chapter summarizes found attack vectors and what assumptions they make, followed by mitigations we propose and conclusions we have drawn.

Summary Plain HTTP is used for updates, logging in and getting device contact information. Communication between app and Camera is done mostly unencrypted via a proprietary UDP protocol. This includes the WiFi credentials for the network the camera is connected to. If mail alarm is set up, the users SMTP credentials are transmitted encrypted, but with a static key that is known. All

this information can be obtained by an attacker if he is able to wiretap communication between device or app and one of the relay servers. It can also be obtained when knowing the device ID and device password.

An attacker who is able to wiretap can also obtain the information necessary to use and control the device. He can initiate a firmware update and if he is active man in the middle somewhere in front of the device, can simulate that a new version is available and deliver a malicious firmware update with arbitrary commands that get executed as root on the device. To generate such a firmware, he adds these commands to the shell script `dhcp.script` that is part of the file system that is delivered via the firmware. It gets copied to the device when updating and is then executed at every boot.

A malicious firmware can also be installed without having man in the middle capabilities, when the attacker knows device ID and device password of his target or is able to wiretap a valid control paket for it. This is done by querying the devices IP settings, setting the same settings with an own IP as DNS server and then initiating an update. This can all be done without knowing the devices IP address. By setting the password of his own device, the attacker can generate valid control pakets for this password and then changes the device ID in the pakets to the target and sends them to the relay server. When the device queries the DNS server for the update domain, an own IP can be answered with a fake update server that serves the malicious firmware.

The paket that is send when refreshing devices in the main view of the app can be recorded and replayed with arbitrary device IDs independently of their device passwords. Because device IDs are just six digits, it is possible to enumerate all possible device IDs in one hour. We found 140741 valid IDs of which each represents an online device.

Every device is delivered with a sticker with the device ID and the standard password `888888` on the bottom. When adding the device to his account, the user is asked to change the password, but can skip doing it. By setting his device to a specific password and pressing on its settings in the app, an attacker can capture a check password paket. By resending it to a relay server with a changed device ID he can check an arbitrary device for his chosen password. By enumerating all online devices, we found 63029 devices that use the default password. Besides the privacy and data harvesting implications we also have all the tools necessary for turning them into a bot net by automatedly installing a malicious firmware on them – even behind router firewalls with out needing to now their IP address.

Mitigations The design of the cloud protocol and infrastructure is fundamentally flawed and requires extensive review and rethought. We will just summarize our most important but not exhaustive mitigation ideas.

The first mitigation would be the use of transport layer security to circumvent information sniffing. This would be HTTPS for the login and updates and Datagram TLS for the UDP protocol. Also, we could imagine a scenario where an attacker impersonates a camera against the server and then gets the control

pakets meant for it. We did not investigate this idea, but if it is not already the case, the device should identify itself utilizing the device password.

Transport security would also circumvent impersonating the update server and delivering a malicious update. In context of defense in depth, firmware images should also be signed by asymmetric cryptography. The private key used to sign the file could be kept at the company and does not need to be on the device for verification.

To mitigate enumerating device IDs, they should be chosen uniformly at random with higher entropy. The initial password should have be individual per device – best uniformly random – and longer. One can think about requiring the user to change it without the possibility to skip after adding it to his account.

How the cloud functionality is designed provides everyone who has access to the central servers with access to all information and devices. This provides high abuse potential and requires a lot of trust. If it will be this way in the future as well, there are many better option for the mail alarm than requiring the user to provide his own SMTP credentials.

As WiFi settings mostly set during the local setup when the device is connected via Ethernet and in the came local network as the smart phone, it is not necessary to transmit the WiFi credentials to the central servers and it should be kept locally.

Further our device and password enumeration showed that there is no (working) rate limiting or monitoring in place, although it is important for infrastructure with high abuse potential.

Conclusion Our research did not require any highly sophisticated tools or secret knowledge. A better implementation would have definitely been possible; by just investing some more time and resources, our attacks could have been prevented. Still, the mitigations are not easily implemented as they require fundamental changes in the cloud protocol, infrastructure and manufacturing process.

Even if the vulnerabilities are patched, there will be other devices vulnerable, because there is a demand for as-cheap-as-possible solutions. In combination with the high numbers of IoT products that are currently and newly will be connected to the Internet, a realistic view on IoT security has to incorporate mass-compromised devices.

Because just blaming the vendors of such insecure devices did not proof as a working solution, we also need to establish reliable DDoS protection – either at the target or at provider level.

5 Demo, Scripts and Tools

5.1 Demo

We implemented two demonstrations to illustrate some of the vulnerabilities we found. They are located in `demo1_sniffing` and `demo2_firmware_update` under the `demo` folder, each with a readme.md markdown file that explains the

usage. The first demo[9] opens a wireless access point that the camera will connect to, then decrypts and displays all SMTP credentials that are transmitted. The

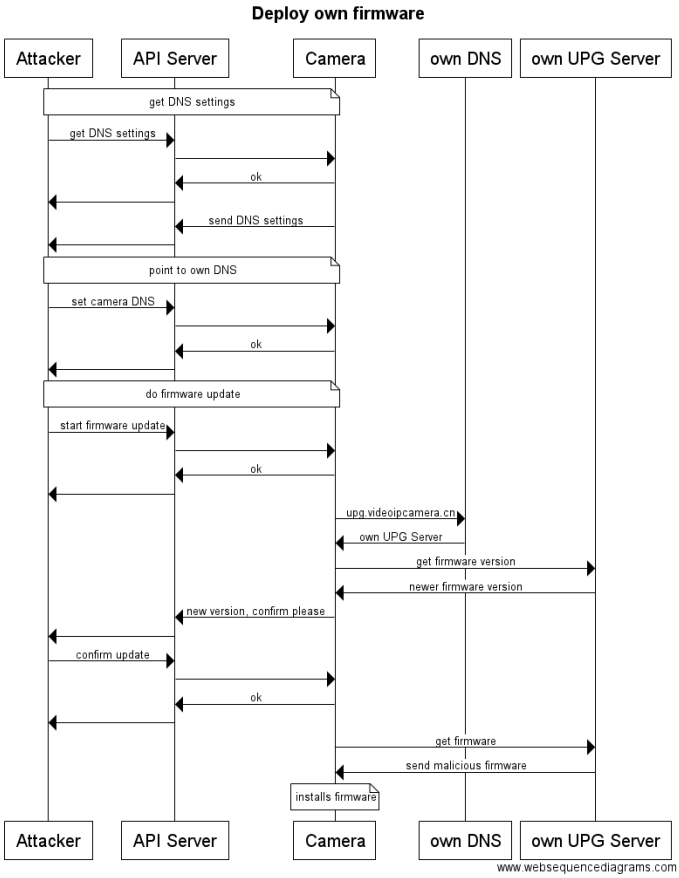

Fig. 15. Process of exploiting a camera by remotely installing own firmware. AP being the attacker who reuses an authentication secret from an own device with the same password.

second demo[10] demonstrates remotely installing a malicious firmware update by just using it's devices ID and pakets created with an own device and the targets password. Fig. 15 illustrates how this process works while **??** shows the network setup that is used. Even though the camera connects to the laptops WiFi in case of the demo, this exact same script would also work with the device being in a network not under the attackers control and without him knowing the IP

[9] Screencast under https://owncloud.hpi.de/index.php/s/Ogr6WwuL4VBcuNI
[10] Screencast under https://owncloud.hpi.de/index.php/s/NwktzfqoeCE0BrQ

address of the device. The only thing necessary would be to host the fake DNS and update server in the Internet. We verified this by trying it out, but went with a local setup for the demonstration to keep it self-contained and future-proof.

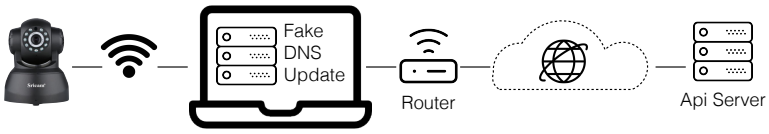

Fig. 16. Network structure for the firmware update demo. The camera is connected to the laptops Wifi to be able to access the fake DNS and update server.

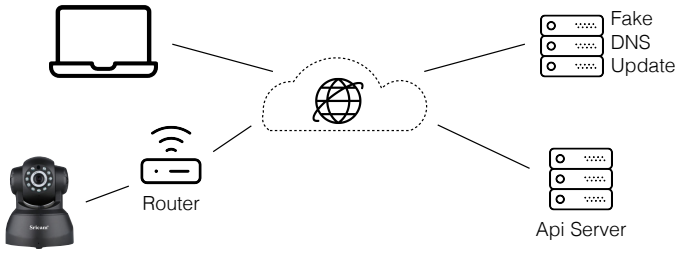

Fig. 17. The network setup how it would be possible as well with the Fake DNS and upgrade server in the Internet.

5.2 Scripts and tools

To give and impression what we used for our investigations, we provide a short list of tools (see Fig. 18) and self written scripts (see Fig. 19) we used.

[12] https://github.com/Crypt0s/FakeDns
[12] https://github.com/buckyroberts/Python-Packet-Sniffer

strings	to find strings, symbols and method names in an executable
xxd	to see a hex view of the bytes in a file
dd	dump filesystem to file, read and write to specific positions in a file
binwalk	analyze binary file to see from what files it is made of
jadx-gui	decompile Android App
Wireshark	sniff all TCP and UDP packets, track connections and filter for bytes in UDP packets
BurpSuite	proxy HTTP requests and edit them on demand to see how the system is behaving
IDA	decompile binary executable, investigate execution flow, search for strings
FakeDNS[11]	for simulating a DNS server by just replacing one domain with our IP and forwarding the rest.
Python-Packet-Sniffer[12]	for wiretapping and analysis the pakets in python. Used in demos and device enumeration.

Fig. 18. Generally available tools that we used for our analysis.

utilities.py	implements some paket and byte conversions, as well as functions to send udp pakets.
mail_alert/helper.py	supplies the exact DES algorithm that is used on the camera and the app.
helpers/packet_sniffer	helps to create proper Ethernet-, IPv4- and UDP-packets.
extract_mail_account.py	checks if a packet contains email credentials by checking for magic bytes. If there are, it extracts them.
sricam_sniffer.py	sets up a WiFi access point, when camera is connected, sniffs udp traffic and extracts email credentials based on "extract_mail_account.py".
sricam_packet_info.py	takes a UDP packet as input and prints all known information of that packet if it belongs to the camera-app-communication.
device_enumeration	folder containing scripts to enumerate devices and check them for specific passwords.
dns_and_http_server.sh	starts a DNS server, that proxies all DNS request and responds a fake server when the Sricam update server is requested. It also starts this python web server for handling the camera's update process.
find_firmware_links.py	enumerates links for possible firmware versions
telnet_dump.py	dumps files from the device by utilizing telnet, cat and base64 encoding

Fig. 19. A selection of scripts that we created for investigation or demonstration purposes. They can be found under their name in the repository. When there were scripts with similar names, we added the directory to prevent confusions.

Bibliography

[1] D. G. A. I. G. M. L. A. e. Bernardo Leal, Luigi Atzori (auth.), *The Internet of Things: 20th Tyrrhenian Workshop on Digital Communications*. Springer-Verlag New York, 1 ed., 2010.

[2] National Intelligence Council, "Disruptive Civil Technologies – six technologies with potential impacts on us interests out to 2025," 04 2008.

[3] L. Atzori, A. Iera, and G. Morabito, "The internet of things: A survey," *Computer networks*, vol. 54, no. 15, pp. 2787–2805, 2010.

[4] E. Vasilomanolakis, J. Daubert, M. Luthra, V. Gazis, A. Wiesmaier, and P. Kikiras, "On the security and privacy of internet of things architectures and systems," in *2015 International Workshop on Secure Internet of Things (SIoT)*, pp. 49–57, IEEE, 2015.

[5] Core Security Technologies, "Hikvision IP Cameras Multiple Vulnerabilities," 08 2013.

[6] David Lodge, "Hacking the Aldi IP CCTV Camera," 10 2015.

[7] Pierre Kim, "Multiple vulnerabilities found in Wireless IP Camera (P2P) WIFICAM cameras and vulnerabilities in custom http server," 03 2017.

[8] Zoltan Balazs, "How I hacked my IP camera, and found this backdoor account ," 09 2015. Updated in October 2016.

[9] J. Granjal, E. Monteiro, and J. S. Silva, "Security for the internet of things: A survey of existing protocols and open research issues," *IEEE Communications Surveys Tutorials*, vol. 17, pp. 1294–1312, thirdquarter 2015.

[10] E. Ronen, C. O'Flynn, A. Shamir, and A. Weingarten, "Iot goes nuclear: Creating a zigbee chain reaction," *IACR Cryptology ePrint Archive*, vol. 2016, p. 1047, 2016.

[11] Brian Krebs, "KrebsOnSecurity Hit With Record DDoS," 09 2016.

[12] Brian Krebs, "Hacked Cameras, DVRs Powered Todays Massive Internet Outage," 10 2016.