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

Supervisor:
Dr. Martin Hell, EIT
Fredrik Larsson

January 25, 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, the older versions do not get the same priority. 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 can have severe consequences, not only for users' privacy, but also for the security of the society.

This thesis is part of a bigger perspective where the goal is to overcome these problems. To offer security and risk analysis of systems and to implement and deploy security patches for them. This thesis focuses on the implementation of a patching mechanism and the deployment of it in a robust way. Also, to consider the security aspects of it such as encryption, signing, PKI but also the need for failure recovery of the system.

Table of Contents

List of Figures

List of Tables

IoT	Internet of Things
M2M	Machine to Machine
IP	Internet Protocol
HTTP	Hypertext Transfer Protocol
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
ICMP	Internet Control Message Protocol
MQTT	Message Queuing Telemetry Transport
CoAP	Constrained Application Protocol
REST	Representational State Transfer
RPL	IPv6 Routing Protocol for Low-Power and Lossy Networks
6LowPAN	IPv6 over Low power Wireless Personal Area Networks
OS	Operating System
RTOS	Real-time Operating System
MCU	Micro Controller Unit
RTSP	Real Time Streaming Protocol
WSN	Wireless Sensor Network
DTLS	Datagram Transport Layer Security
SSL	Secure Socket Layer
AES	Advanced Encryption Standard
ECC	Elliptic Curve Cryptography
DSA	Digital Signature Algorithm
ECDSA	Elliptic Curve Digital Signature Algorithm
DES	Data Encryption Standard
MAC	Message Authentication Code
CA	Certificate Authority
PKI	Public Key Infrastructure
DoS	Denial of Service
DM	Device Management
MDM	Mobile Device Management
OTA	Over the Air

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 we see a tremendous increase in the number of devices connected to the Internet. Many companies estimates it to reach a number of about 50 billion connected devices by the year of 2020 [5]. With such a remarkable amount of devices we 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 any vulnerabilities are found. Another problem is all the older software versions used in products that are no longer produced or maintained by companies. Products that are still widely in use. All these products can pose a threat to the society and the user privacy if the software is outdated and exposed to the Internet. It is of uttermost importance to find viable ways to increase the security in all IoT devices, even the older ones. 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 and for 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 needs to be evaluated for a specific device or system, hence enough information about the system is needed in order to take appropriate action. Planning is needed to decide when the patch should be implemented and if all devices should be updated or just a subset. Furthermore, a patch needs to be written 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. This process is summarized in figure 1.1. It can be utilized in any area of software patch management, where our focus will be on IoT devices and on The Company's products.

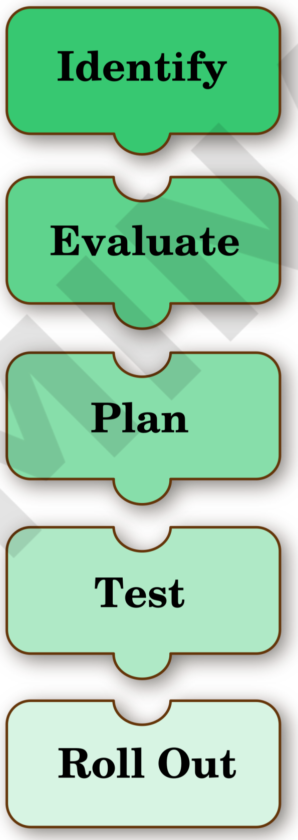


Figure 1.1: A five step model for secure updates

1.1 Purpose and Goals

The purpose for this Master's 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.

Some of the sub goals to this are to investigate the current state of the art methods for updating and for software management and also to investigate the contemporary software used in IoT devices. Another goal is to implement a proof of concept, with a working and secure update mechanism.

Much of this Master's thesis' work is done at The Company, and it is a collaboration between Lunds Tekniska Högskola (LTH) and The Company.

1.2 About LTH

Lunds Tekniska Högskola is a faculty of the Lund University. With almost 10 000 students it is one of the biggest technical higher degree schools in Sweden. LTH and Lund University has been listed in the top 100 University rankings for several years [37] [10].

Chapter 2

Background

This chapter introduces some of the fundamental concepts for understanding the Internet of Things and security related issues.

2.1 Internet of Things

Internet of Things is the trending term for objects or “things” equipped with processors and sensors, begin able to be aware of its 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 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 there are, the larger the “playground” is for adversaries. This opens up a whole new world and it leads to almost endless possibilities, and the impacts can be severe. This puts serious pressure on the security mechanisms in IoT. Many times, the security is not even considered when a product is being developed. The main goal is to make it work, for as low cost as possible. Implementing security is both time consuming and makes the code bigger and more complex which is undesirable, especially in IoT devices. As security is becoming such a crucial part of the software development, one can not afford to not have it implemented.

The IoT devices are to be integrated into an already existing ecosystem - the Internet.

2.2 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, they still have to use the standard IP stack in some sense to be able to communicate with other connected devices. Some sort of adaptation is needed. For this reason, some 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 widely used, new standard lightweight protocols are explained below. Figure 2.1 shows the differences and similarities between the original TCP/IP stack and the lightweight focused IoT stack.

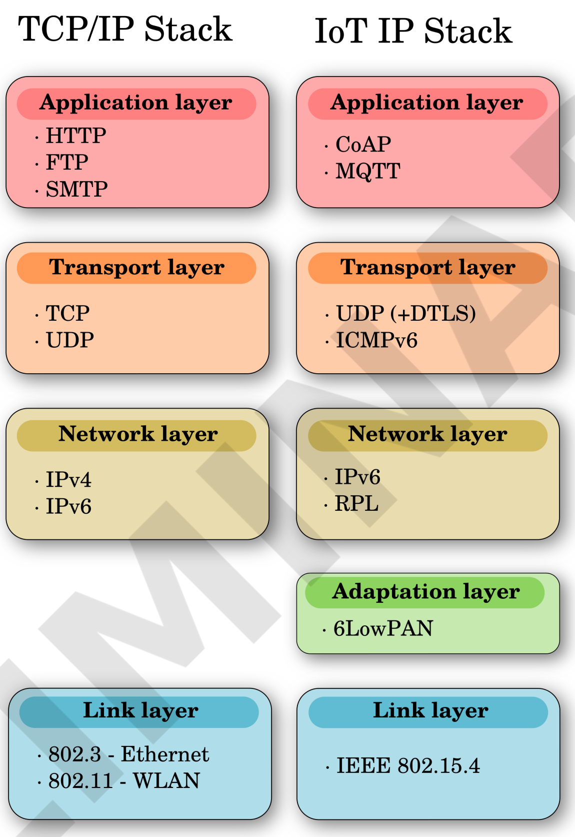


Figure 2.1: The TCP/IP and the IoT IP stack

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 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 this, CoAP provides security in the form of DTLS which with default parameters is equal to a 3072-bits RSA key.

UDP[2] is a stateless protocol since delivery of packets is not guaranteed. Unlike TCP, packets can arrive in any order in UDP. This makes UDP more of a lightweight protocol compared to TCP, and thus more suited for IoT devices.

[1]`http://coap.technology/`
[2]`https://www.ietf.org/rfc/rfc768.txt`

RPL[1] is a 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 has different security implementations.

6LowPAN[2,3] is an adaptation protocol that makes it possible to send and recieve 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. ECC provides the same level of security as of RSA or AES but with a smaller key size.

IEEE 802.15.4[4] is a standard for resource constrained devices on the physical layer and the MAC layer. It can build up a wireless embedded Internet together with 6LowPAN. It is also basis for other standards such as ZigBee[5].

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.

2.3 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. Below we introduce some open-source operating systems commonly used in IoT devices. In Table 2.1 the OSs are put against each other to point out differences and similarities [3].

Contiki[6] is a real time operating system (RTOS), written in C, originally developed for use 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

[1]`https://tools.ietf.org/html/rfc6550`
[3]`https://tools.ietf.org/html/rfc4944`
[3]`https://tools.ietf.org/html/rfc6282`
[4]`http://www.ieee802.org/15/pub/TG4.html`
[5]`http://www.zigbee.org/`
[6]`http://www.contiki-os.org/`

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 [8].

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 [30]. 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 [3]. Contiki is a multi-threaded OS and offers for example a UNIX-like shell. Communication security is provided via ContikiSec [34], a protocol for network layer security.

TinyOS[1] is an operating system written in NesC, 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. After a while, a callback method is run to check if the previous method call is done. 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) protocol and one that they created themselves 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. TinyOS's first implementation of 6LowPAN was called 6lowpancli but drawbacks led to a second implementation, BLIP (Berkeley Low-Power IP stack) [30].

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 [3] including the whole Mica family, TelosB, IRIS, XYZ etc. The communication security in TinyOS is provided via TinySec [36], which is a link-layer security architecture for WSNs.

RIOT[2] 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 many devices ranging from 8-bit MCUs such as the Arduino 2560, to 32-bits MCUs such as ARM, and it is partial POSIX-compliant. The constraints on hardware are small, thus making RIOT a suitable option in small connected devices.

LiteOS[3] is an OS written in LiteC++ with the goal to provide a Unix-like environment for 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 [12]. LiteOS supports multi-

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

[2] `http://www.riot-os.org/`

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

	Contiki	TinyOS	RIOT	LiteOS
Publication	2000	2004	2013	2008
Static / dynamic	Dynamic	Static	Dynamic	Dynamic
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, XYZ, IRIS, Tinynode, Eyes, Shimmer	Arduino, MSP430, ARM	MicaZ, IRIS, AVR

Table 2.1: Comparison of the OSs

threading and dynamic loading and can be run on the MicaZ and IRIS sensor nodes as well as on AVR [3].

2.4 Wireless Sensor Networks

Wireless Sensor Networks (WSNs) is an emerging technology, consisting of a group of small (IoT-)devices or sensors (called nodes), often connected to each other in a mesh network. The nodes are normally very resource restricted and usually does not have a connection to the Internet by themselves. For this reason, a gateway is used as a master for all the nodes in the network. It handles the 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, meaning that packets are sometimes lost during transmission. The loss rate is increasing with increased range [1], which is why it is important that the nodes can communicate with each other. 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 called dissemination protocols are used [2]. These protocols define the transmission between nodes, and are explained more in section 2.8.

WSNs are usually deployed in areas that are hard to access [15] [11]. Because of this, an over-the-air update mechanisms is often a requirement. Some examples of WSN applications are wildlife monitoring, military command, intelligent communication, critical infrastructure observation, smart homes, distributed robotics, traffic monitoring etc [14]. 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 **??**.

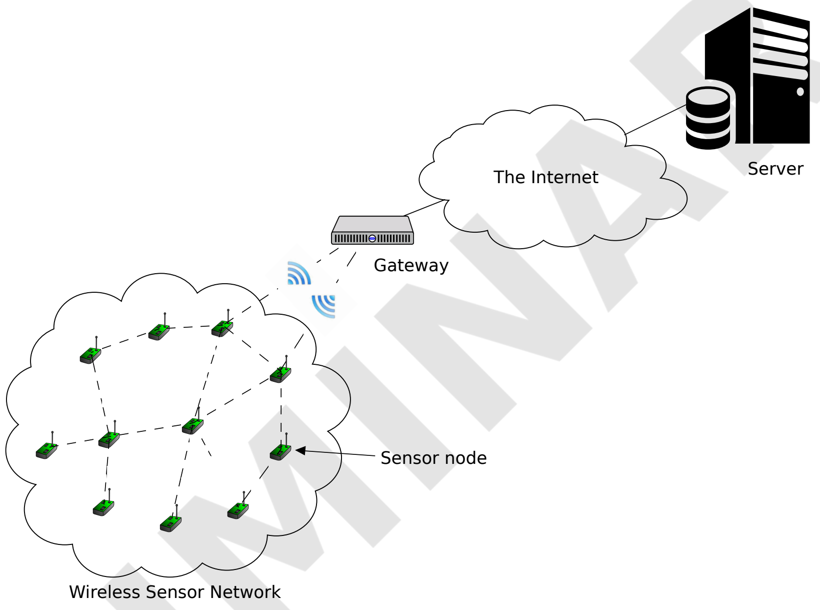


Figure 2.2: Typical WSN model

2.5 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, we introduce basic theory on security. For a more detailed description regarding security and cryptography, consult the *Handbook of Applied Cryptography* [21].

2.5.1 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.

2.5.2 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 since the keys are mathematically linked together. A message encrypted with the public key can only be decrypted with the corresponding private key.

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

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 is 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 key are derived. The strength of RSA lies in the difficulty of factoring large prime numbers[2].

ECC (Elliptic curve cryptography [23]) 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].

2.5.3 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 and that the sender can not, 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.

2.5.4 Cryptographic Hash Functions

A cryptographic hash function is a non-invertible function that maps an input of any size to a fixed-size output, known as a hash. Cryptographic hash functions is commonly used in digital signatures and message authentication codes (MACs)

[1]`http://searchsecurity.techtarget.com/answer/What-are-new-and-commonly-used-public-key-cryptography-algor`
[2]`https://en.wikipedia.org/wiki/Integer_factorization`
[3]`https://en.wikipedia.org/wiki/Discrete_logarithm`

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 $H(m_1) = H(m_2)$.
- 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.

2.6 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 form of a digital certificate, see below. This certificate is digitally signed by an authority known as a Certificate Authority (CA). The user's identity can 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.

2.6.1 Digital Certificates

A digital certificate, or public key certificate, contains information about a public key and the corresponding user, thereby providing the ownership of the public key.

2.7 Security in Wireless Sensor Networks

Because of a wide range of deployment in many different scenarios, especially in hostile environments with intelligent opposition, the security in WSNs is essential. Many sensor nodes may be deployed in unattended environments, which makes 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 [14]. The main security goals are still the same as for conventional security though - to achieve message confidentiality, integrity, authenticity, availability and non-repudiation.

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

2.7.1 Attacks and security in WSNs

In WSNs, large volumes of information is transferred between the nodes with sometimes little thought of security. 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 forms 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 can 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 often seen as the most important security issue [4]. To achieve data confidentiality and protect against eavesdropping etc., the standard way is to encrypt the sent data with a secret key. But only confidentiality does not make the data safe. It is 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 infers 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 needs careful consideration. Traditionally, all asymmetric cryptography techniques were seen as too intense for sensor networks [4]. This is still true in the general case, but it is shown that it can be done with the right selection of algorithms [33]. RSA and even Elliptic curves [29] 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 [4].

2.8 Updates

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 employed 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 introduces a risk that something could stop working.

We have to distinguish between a regular update and a security update. A security update, which is what we focus on, is not supposed to touch any features. Its only purpose is to fix security vulnerabilities and to make the code more secure. Companies might be more interested in those kind of updates, and even more important, if the code are based on open source, the security updates could have already been tested by the community.

2.8.1 Dissemination

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 [2]. One common example is Deluge [6] which is developed to propagate large amounts of data, especially dissemination of 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.

The protocols have the advantage that the nodes in the network can act both like 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 unnecessary bandwidth.

A typical pattern for the nodes is called the `advertisement pattern` (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` (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 WSNs [2].

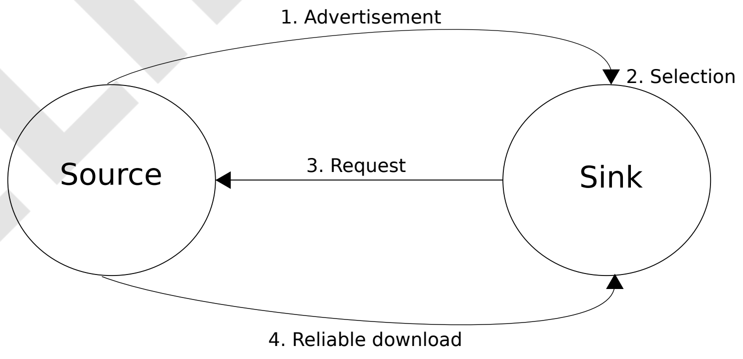


Figure 2.3: Advertisement Pattern

2.8.2 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 track of program states and code. Today, operating systems and programming languages

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

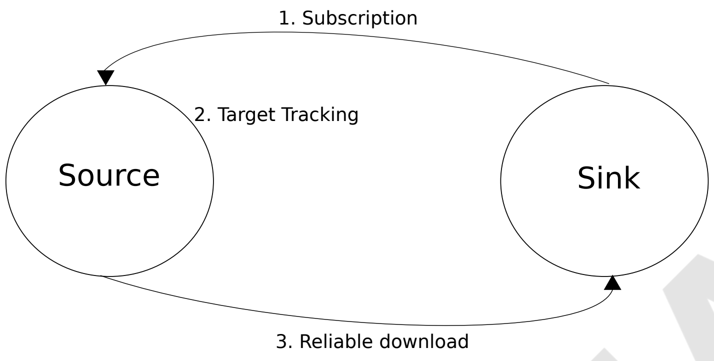


Figure 2.4: Subscription Pattern

are typically not designed with DSU in mind[1]. To allow for DSU it is common to implement specialised compliers to preserve the semantics of the program and to make it possible to dynamically update it.

2.8.3 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. That is because the delta patch also contains other information than just the code. It also has to keep track of the differences and where they are. But one advantage with a delta patch is that when using compression algorithms on the code, the patch can become very small, smaller than the original binary because of recurring information.

2.8.4 Major/Minor Updates

There is a difference between a major and a minor update. When a major update where many parts of the code is changed is about to be rolled out, the best option might not be a delta patch but a "normal" update. The many changes in the code will make the delta become infeasibly large. But for minor updates where just small parts of the code is changed, the delta patch would be a better option in most cases.

[1]`https://en.wikipedia.org/wiki/Dynamic_software_updating`

2.8.5 Over the Air

2.9 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 recieve data. The protocol features include support for device monitoring, configuring and updating.

2.10 Risk Analysis

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" [35]. Another one 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" [20].

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. Doing 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 .

[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

Sometimes these tools does not give a completely accurate result, thus making it less likely for the company to make an ideal decision.

The risks have to consider the impact of not applying a security patch but also the impact of applying it. First of all, a security hole needs to be identified. Then it has to be evaluated, i.e. how does this vulnerability affect the systems? After the evaluation and the decision to implement a patch the planning phase begins. After that, the actual implementation and roll out of the patch can be done.

Chapter 3

Vulnerability Assessment

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

– Adi Shamir

Vulnerabilities in The Company's products is a major concern since it allows an attacker to gain access to the video stream. The cameras are also 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 its impacts.

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

Identify
Evaluate

Figure 3.1: The two main parts in 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 [18] feeding the database with information. Even though the CVEs cover lots of vulnerabilities, they are not complete. In a report

[27], 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] provides detailed information on vulnerabilities and mailing lists for subscription. It is also possible to follow open source projects directly, GitHub[8], SourceForge[9] etc.

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], Coverity[11], Nessus[12] among others.

[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`

Identification at The Company

In The Company's case, 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 [31], 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 does 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, The Company should utilize several of the above information sources:

- CVE Details[1] offers RSS feeds and JSON URLs 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 exists. 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 [31] resulting in a lack of awareness of the flaws. Thus, scanning repository logs as a complement to 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 [24], 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, OSVDB etc. 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 can be used. In [38] it is shown that bug reports are easily mislabelled as non security related when they in fact are. They have also developed a statistical text mining model to identify the mislabelled bug reports. By using such tool(s), The Company could scan all reports, security and non-security, to find those mislabelled reports in an automated way. The same techniques can 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 listed below:

[1]https://www.riskbasedsecurity.com/vulndb/

```
#!/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's lifetime [7].

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, can be used in order to evaluate a system. One of the most common ones is the 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:

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

[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 [26].

CWSS[1] is a scoring system that is quite similar to CVSS. To calculate a vulnerability 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 [19].

A framework called CWRAF[2] provides a way for companies to apply CWSS and customize it on the CWEs most relevant to the company's 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 partly specific for a system. The problem with the results being too general still exists. Even though CWRAF knows the company domain, the result might be misleading compared to the reality.

Evaluation at The Company

When a software vulnerability has been found, The Company begins a process of meetings. 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 board 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 The Company's systems, a vulnerability report specifically for The Company comes as a result from the evaluation process. The amount of man-hours spent can vary widely depending on the effect of the vulnerability. A significant amount of time can be saved by setting up guidelines for this process and to specify roles and responsibilities for it.

The Company uses vulnerability scanners, like Nessus, in order to find any weaknesses. Nessus scans a target and ranks any 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 consequences of an exploit on the system into account. For the Heartbleed bug (CVE-2014-0160)[1], 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 is 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, The Company 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.
- Buy 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] (a technical and business context) have to be defined, either to create a new one or choose from existing ones. Weightings for specific business cases has to be added for the vignette. Then analysis tools is 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 needs to be tracked from many different sources and combined into one result. Unfortunately, the available tools does not give a specific result with system configurations in mind.

A future step to take could be to implement machine learning algorithms as in [26]. 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

Case Study - Evaluation

In order to perform adequate identification and evaluation of vulnerabilities, we need to obtain relevant information of the product or system. 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 root access to the camera and the video stream.

This attack is mainly dependent on system configurations, i.e. if this software is in use and what version it has. 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. The Company'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.

```
PACKAGE NAME: openssl
PACKAGE VERSION: 1.0.2d
RECIPE NAME: openssl
LICENSE: openssl
```

[1] http://heartbleed.com/

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 The Company's products supported the SSL v3 protocol which made them vulnerable. The given impact analysis by The Company 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 some 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 the system environment in which cameras accessed from the Internet were more exposed.

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`

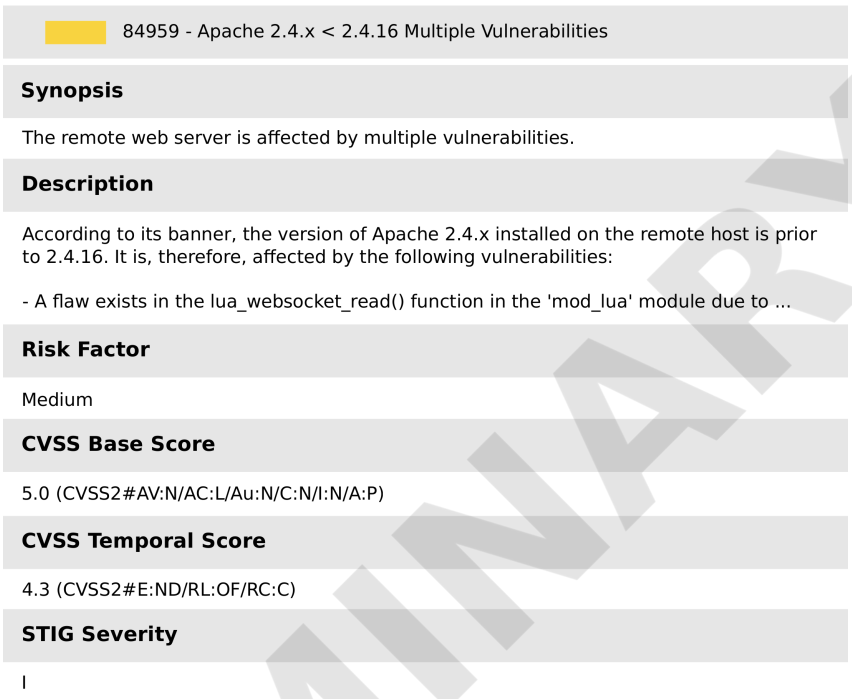
```
> httpd -M
Loaded Modules:
 core_module (static)
 so_module (static)
 http_module (static)
 suexec_module (static)
 mime_module (shared)
 mpm_worker_module (shared)
 unixd_module (shared)
 alias_module (shared)
 rewrite_module (shared)
 cgid_module (shared)
 log_config_module (shared)
 setenvif_module (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)
 axsyslog_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 on a web application. The user privileges are inherited so if the user is root, the attacker will also be identified as root. 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 The Company since only a few, estimated to 5 %, cameras are accessed

[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


84959 - Apache 2.4.x < 2.4.16 Multiple Vulnerabilities

Synopsis

The remote web server is affected by multiple vulnerabilities.

Description

According to its banner, the version of Apache 2.4.x installed on the remote host is prior to 2.4.16. It is, therefore, affected by the following vulnerabilities:

- A flaw exists in the lua_websocket_read() function in the 'mod_lua' module due to ...

Risk Factor

Medium

CVSS Base Score

5.0 (CVSS2#AV:N/AC:L/Au:N/C:N/I:N/A:P)

CVSS Temporal Score

4.3 (CVSS2#E:ND/RL:OF/RC:C)

STIG Severity

I

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

through 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.

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 was not designed for the rapid development of connected devices in mind, such as IoT products where interconnectivity is the focus [13]. 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 is the system configuration and its environment. Some examples, as found in the case study, of the factors are show 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 of 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 stored as a cookie in the browser. If the product further is vulnerable to XSS, the attacker may read the cookie information 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 via text mining and evaluation via impact analysis.

The conclusion from this case study shows that with some knowledge about the system, a more fine-grained evaluation can be achieved. The process can be automated to a larger extent using machine learning. The more complex a system is, the more extensive the data gathering must be.

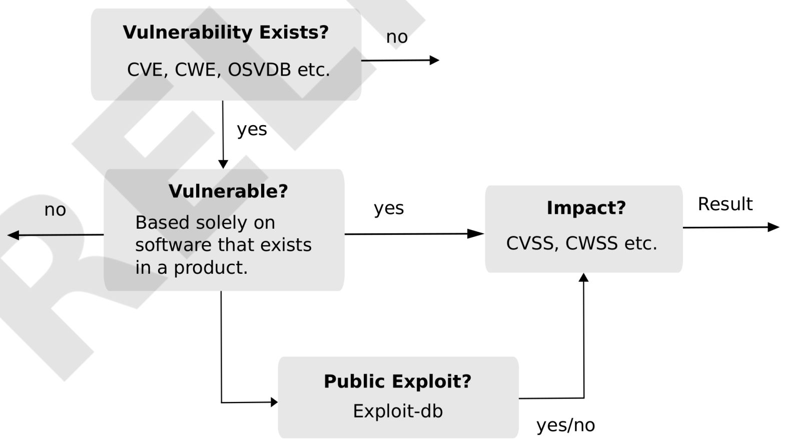


Figure 4.2: A typical method for identification and evaluation. They are very general and the output is based on a high-level description of a system.

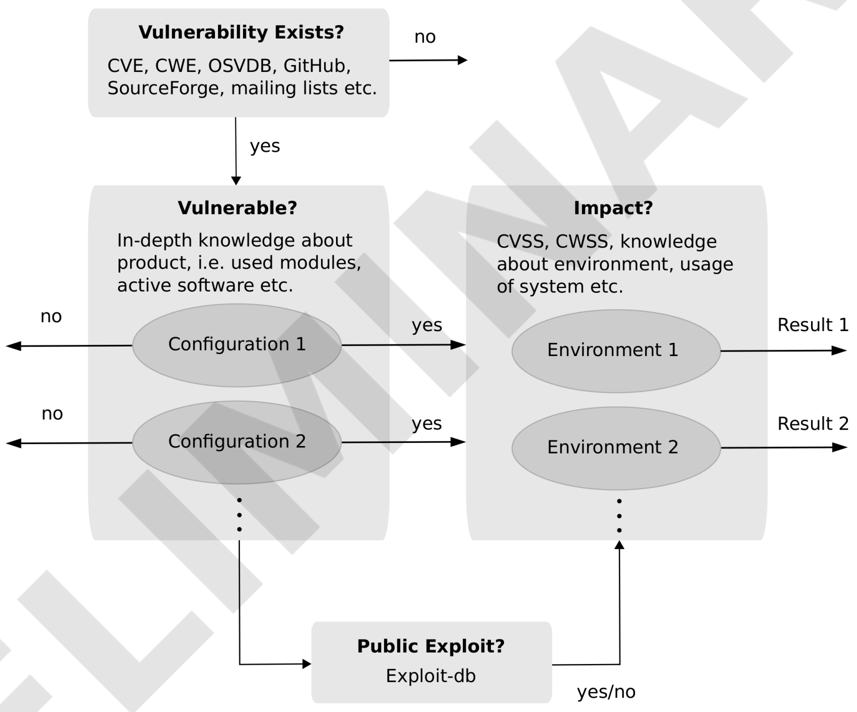

Vulnerability Exists?
CVE, CWE, OSVDB, GitHub, SourceForge, mailing lists etc.
no
yes
Vulnerable?
In-depth knowledge about product, i.e. used modules, active software etc.
no
Configuration 1
yes
no
Configuration 2
yes
Impact?
CVSS, CWSS, knowledge about environment, usage of system etc.
Environment 1
Result 1
Environment 2
Result 2
Public Exploit?
Exploit-db
yes/no

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: Summary of how configuration and environment affect the applicability of an attack. A product is marked green if it is not vulnerable. It is marked red if vulnerable

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, 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

We are using an Android product, the Nexus 5X, and a Chromebook as references due to their already working update mechanisms. These products are then compared to small IoT devices in WSNs and also to The Company'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 stepping stone 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 product using Android is the smartphone and tablet. They are stand-alone devices used in different environments, such as homes, offices and in the 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 [9] [17] [25].

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, specifications and security for the Nexus 5X is 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/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 way to change the contents is during an OTA update. 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-write. 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 certificates is used to check the cryptographic signature.
- 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.
- 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

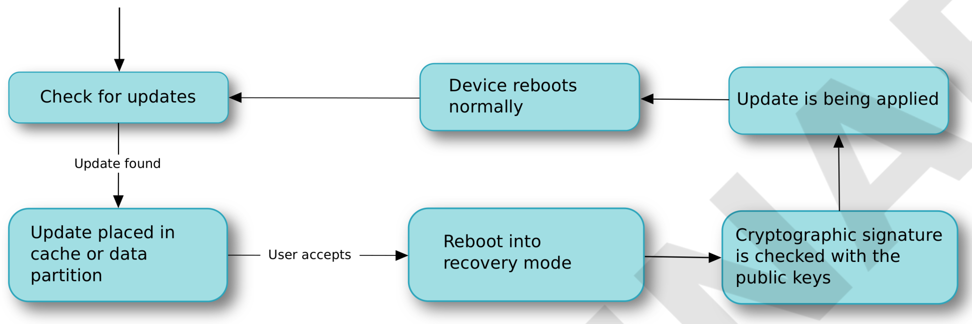

Check for updates
Device reboots normally
Update is being applied
Update found
Update placed in cache or data partition
User accepts
Reboot into recovery mode
Cryptographic signature is checked with the public keys

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 type of computer, running Chrome OS, developed by Google. Chrome OS is based on one of the most used software applications – the browser. Below, we describe the update process and other parts related to it, such as the file system.

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, 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] we find the update process flow, here compiled as 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. 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.

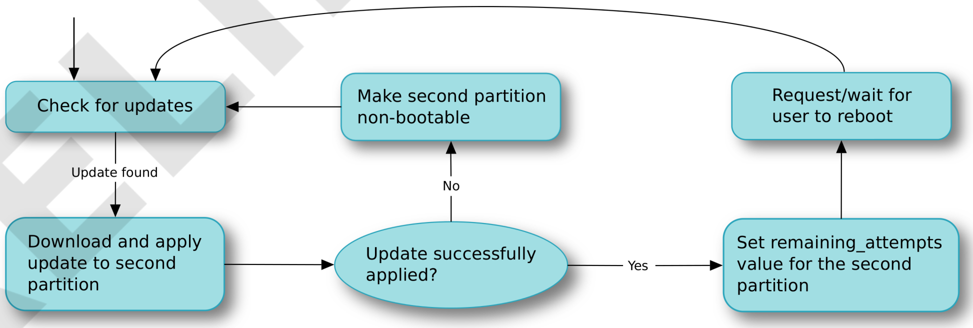

Check for updates
Make second partition non-bootable
Request/wait for user to reboot
Update found
No
Download and apply update to second partition
Update successfully applied?
Yes
Set remaining_attempts value for the second partition

Figure 5.2: Update process in Chrome OS

Security

All updates are downloaded over HTTPS which means that the communication is encrypted, thus it is less likely that the updates have been tampered with on the

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

way to the user. A checksum and a signature is also sent along with the update.

5.2 Targeted Use Cases

The following use cases are the focus of this Master's 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 The Company Use Cases

The Company has a variety of products used in many different scenarios. Here, we present two use cases and discuss the demands on the cameras, etc.

A public transport company

A customer of The Company, a public transport company, recently installed The Company's 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 cite and no direct connection to the Internet for the cameras. This implies that the system is not reachable from outside and that found software vulnerabilities are not of great impact. 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. That is because the trains needs to be out of service for as little as possible as that costs money.
- The cameras are updated per train, by the maintenance technicians.
- It takes about ten minutes to update the cameras in a train once the technician is connected to the train network.
- If it is not essential to update, it is avoided where possible because it takes effort.
- "Never change a running system."

Today the updates are installed manually by the technicians. Instead of this, a fully automated update procedure would be of interest. Primarily because that would lower the out of service time. But, a fully automated update procedures also introduces problems. The whole system has to be considered and everything needs to support the latest firmware. 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

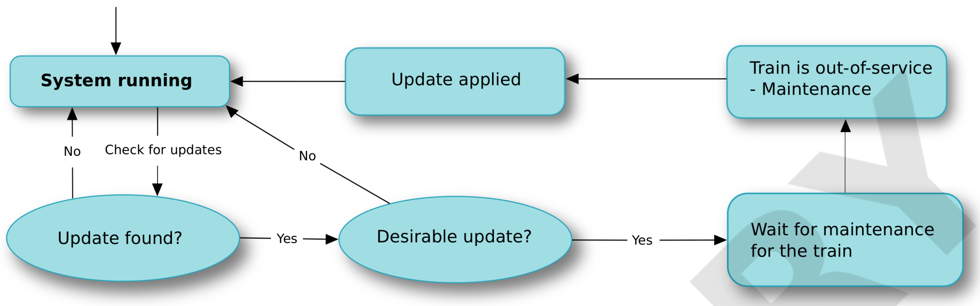


Figure 5.3: Update process for the public transport company

Security

Security is considered to be important, but that statement is not well supported in real life. For example, no https or 802.1x are used and the cameras have their default passwords. If these basic things are not taken into account, then any security update would not make much of a difference anyway.

An Enterprise Retailer

A big customer of The Company have 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 things 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 platforms 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 does 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 considered, 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 5.4

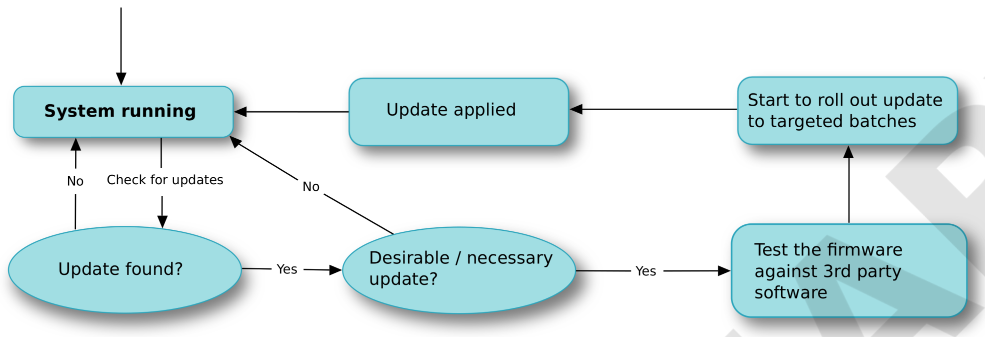

System running
Update applied
Start to roll out update to targeted batches
No
Check for updates
No
Update found?
Yes
Desirable / necessary update?
Yes
Test the firmware against 3rd party software

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. But the fact that this process is so time consuming is a vulnerability in itself. If the update process can be boosted, many threats can be reduced, at least in time.

5.2.2 WSN

A Wireless Sensor Network is a network of usually small devices or sensors with wireless communication capabilities. These devices are called “nodes”. The Mica2 Mote is an example of a WSN node. The word “mote” comes from the old English word “mot” which means a 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, RH, 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. So 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 consists of resource restricted devices, it is important to have as low data transmission as possible. For this purpose, delta-compressed updates is essential, just like in the Chromebook OS case.

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

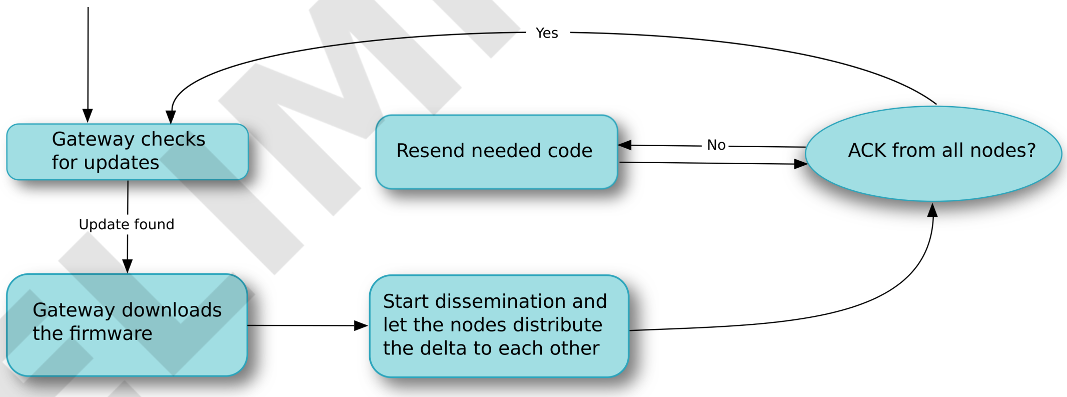

Yes
Gateway checks for updates
Resend needed code
No
ACK from all nodes?
Update found
Gateway downloads the firmware
Start dissemination and let the nodes distribute the delta to each other

Figure 5.5: Update process in WSNs

Security

In [11], the security in WSNs is discussed and said that attacks in WSNs are similar to those in wired networks. It also says 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.

5.3 Comparison

Table 5.1 compares the different use cases in terms of protocols and gives an overview of the differences and similarities.

	OS			**Protocols & Network**										
	Contiki	TinyOS	Linux	WiFi	3G/LTE	Radio	HTTP	CoAP	TCP	UDP	IPv6	RPL	6LowPAN	802.15.4
Yanzi Led	✓			✓				✓	✓	✓	✓	✓	✓	✓
Mica2 Mote (WSN)		✓				✓		✓		✓	✓	✓	✓	✓
Android OS			✓	✓	✓		✓		✓	✓	✓			
Chrome OS			✓	✓	✓		✓		✓	✓	✓			
The Company's cameras			✓	✓			✓		✓					

Table 5.1: Use case comparison

Chapter 6

Deployment

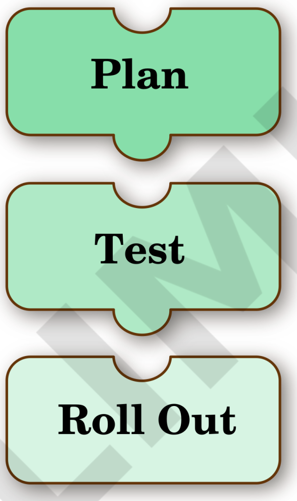


Figure 6.1: The three main parts in patch deployment.

6.1 Planning

The planning phase of the assessment is initialized if a vulnerability at this point was considered important enough to be patched. Questions like when and how to deploy the patch are raised here. 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 phase is not the focus of this thesis and will thus not be explored further.

6.2 Testing

This phase includes both assembling and testing of the patch. If a patch already exists, the assembly could very simple by just applying it to the device. It might also involve complete rebuilds of the image to make the patch applicable. To find an already existing patch is often the case for popular open source software. For proprietary software and for some open source software, the patch needs to be built.

When the patch has been assembled, it is tested in virtual test environments similar to the real environment. A test process should start with a verification of of the patch's source and integrity and it should contain some form of a digital signature or a checksum [22]. This ensures that the patch is valid and not altered with. The mechanisms in the test process can vary much from one company to the next. It is dependent on system criticality and availability, how severe the patch is, and the available resources.

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

6.3 Proposed Solution

Before we continue to the roll out phase we consider the deployment use cases in chapter 5 again. By combining desired functionalities from the use case study, we have proposed a solution that is generic and usable across many different devices, from the smallest IoT sensor to computers with high computational power.

Presented below is a proposed update process combining the desired functionalities from existing processes, and ensuring compatibility for multiple platforms. Two different processes are included, one fully automatic and one semi-automatic. One should not omit the other. For some systems it is not desirable with a fully automated process e.g. for a security camera system, while for some it might be, e.g. a sensor network.

Both processes include a trusted source distributing the updates. That could also be done by the company itself. Figure 6.2 presents the semi-automatic solution and figure 6.3 the fully automatic solution.

This proposed solution uses 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 function. Figure 6.4 visualizes what happens on the two partitions during a patch.

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 what patch to deploy. This and other considerations are omitted in order to keep it as generic as possible.

Pros and Cons with Some of the Design Choices

Two partitions

Pros: Being able to apply updates while the system is running (but not on the running partition), as well as to stack updates. Also gives the possibility

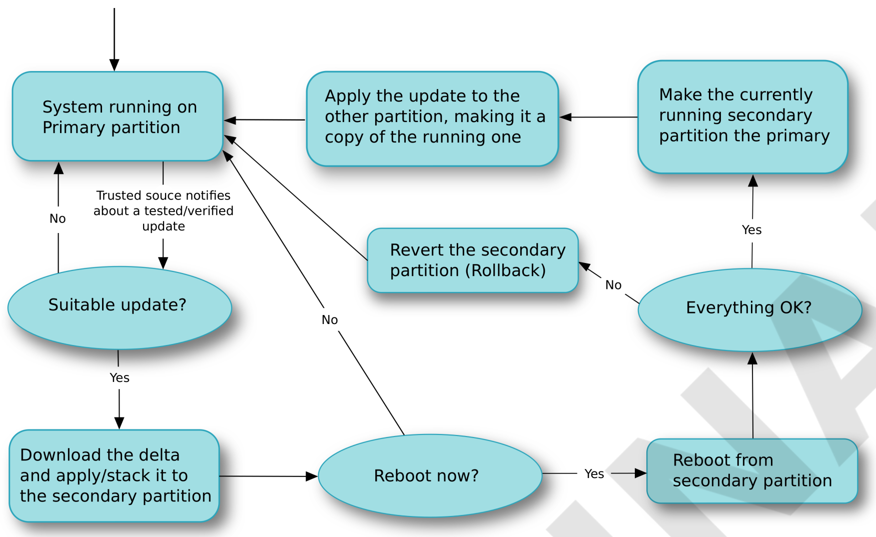

System running on Primary partition
Apply the update to the other partition, making it a copy of the running one
Make the currently running secondary partition the primary
Trusted souce notifies about a tested/verified update
No
Yes
Revert the secondary partition (Rollback)
No
Suitable update?
No
Everything OK?
Yes
Download the delta and apply/stack it to the secondary partition
Reboot now?
Yes
Reboot from secondary partition

Figure 6.2: Ideal Update Process - Semi-Automatic

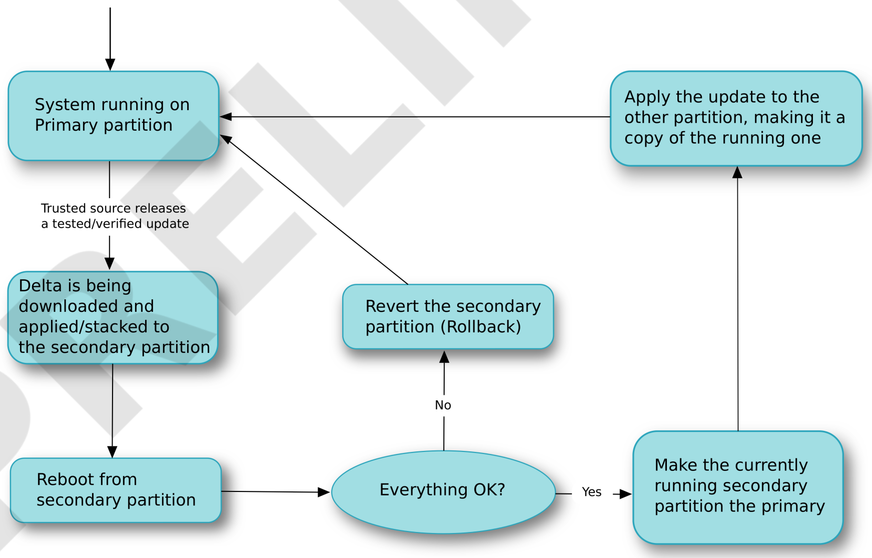

System running on Primary partition
Apply the update to the other partition, making it a copy of the running one
Trusted source releases a tested/verified update
Delta is being downloaded and applied/stacked to the secondary partition
Revert the secondary partition (Rollback)
No
Reboot from secondary partition
Everything OK?
Yes
Make the currently running secondary partition the primary

Figure 6.3: Ideal Update Process - Fully Automatic

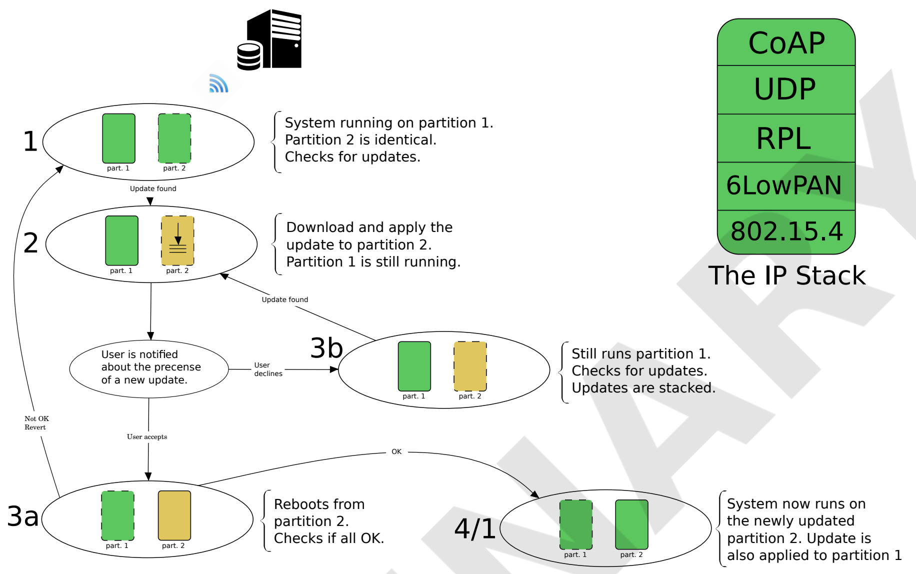


Figure 6.4: Ideal Update Process (Graphical Illustration)

to do a roll-back
Cons: Requires more hard drive space. Requires a reboot to run the newly updated partition.

Deltas

Pros: Less code transmitted over the network.
Cons: Harder to handle than a full firmware update. Needs tables for version management.

User notification

Pros: Gives the user control of what is updated and when.
Cons: It is not installed instantly, which might be a problem for crucial security updates.

Chapter 7

Roll Out

When a patch has been fully tested and verified in test environments, the roll out can begin (see 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 The Company, 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 binary to detect any errors in the transmitted data.

To send a full firmware every time a camera is being updated means an unnecessarily large amount of data being transmitted over the network. However, when an update contains many changes, a full firmware update could still be the best way to do it. When it is a matter of minor bug fixes and small changes, a patch will result in a much smaller amount of data being transmitted. For The Company, with devices with considerably large amount of storage and with a good TCP connection, the gain in decreasing the number of bits transmitted is marginal. Still, a mechanism for patching and even more so, more added security features, is desirable for The Company. Not only because of less transmitted bits, but also to be able to increase the update frequency of the cameras. For smaller IoT devices, a patching mechanism is even more important because of the limited storage and bandwidth they possess.

7.1 Protocols

7.2 Operating Systems

Our patch program is written for Linux because The Company's cameras are running Linux. It is convenient because many known algorithms 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 Chapter 2. There already exist libraries ported to for example Contiki, such as the bsdiff delta encoding algorithm and the LZ77 (de)compression algorithm [32].

Those two algorithms are also shown to be the best combination of compression and delta encoding in terms of performance [16].

7.3 Security

7.4 Distribution

7.4.1 Dissemination

7.5 Architectural considerations

The architecture of the high level hardware must be carefully considered. One of the most important goals to achieve is to have roll back capabilities - if something goes wrong, we can always go back to a working version.

7.5.1 OverlayFS

OverlayFS is a file system that allows files to be written to a “shadow” file. It makes several file systems appear as one when mounted. It was introduced in Linux in kernal version 3.18 and is now a part of the mainline kernel. An overlay-filesystem is the concept of one filesystem placed on top of another, called the upper and lower filesystems[1]. If a file exists in both filesystems, you only see the file from the upper filesystem while the lower file is hidden. If the object instead is a directory then a merged directory is created. When a lookup is requested in a merged directory, the lookup is performed in each actual directory. The lower filesystem can be of any kind supported by Linux, writable or not. It can even be an overlay filesystem in itself. The upper filesystem is usually writable.

A desirable feature for this project is that this file system architecture allows the OS to roll back to a previous version by simply not reading the “shadow” file, in case the update crashed or similar.

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

7.6 Our Program / Deploy a Patch ?????

We have implemented a program to patch binaries on The Company’s cameras in a secure way. The current solution uses a server/client based approach. The server is run on a host, e.g. a server of The Company, containing binaries to updated programs used in the cameras. The client is run on the cameras for example once

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

every hour to check if there is a new version of a program on the server. If there is a program on the server that is not on the camera, the program is downloaded. If there is a new version of an existing program, the patch is downloaded - A “diff” between the currently running program and the updated version of it.

The patch data that is being sent over the network contains a header with information about the patch. The structure of the whole packet looks like this:

```
Packet:
 ------------------------------
|  Header  |  Compressed data  |
 ------------------------------
```

The structure of the header is seen below. It consists of 5 fields:

```
Header:

 1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-------+-------------------------------------------------------+
| dsize |                     Path                              |
+-+-+---+-------------------------------------------------------+
|F|L|                  ECDSA Signature
+-+-+-----------------------------------------------------------

-----------------------------------------------------------------

-----------------------------------------------------------------

------------------------------+----------------------------------
                              |
------------------------------+
```

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

Path
The path to the file being patched or replaced.

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

- bit 7 to 3: For future use.
- bit 2: data is compressed.

- bit 1: data contains a full binary.
- bit 0: data contains a patch.

Signature length (L)
The length of the ECDSA signature.

ECDSA Signature
The ECDSA signature value. The maximum signature length is 141 bytes, but the actual signature length can be much smaller. This is due to a non-deterministic function for calculating the signature.

When a patch is to be distributed, the server runs the diff program. The server side needs to have both the old version and the new version of the program in order to create the patch. The diff program builds the header for the packet. It also calculates a checksum for the whole packet and adds its signature to it, verifying the authenticity of the sender (The Company) 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.

The client checks for updates, receives a patch and starts to read the header. If anything has happened with the packet during the transmission, the client will notice it and discard the packet. This includes events such as lost data during transmission, someone who altered the packet on its way, someone who sent another packet trying to make it look like a packet from The Company etc. The signature makes sure all those events are prevented.

The confidentiality of the packet can not be ensured since we decided not to encrypt the data because we only focus on open source software. The data is public anyway. If also proprietary code is to be sent, confidentiality can be easily ensured in the future by adding encryption such as AES.

When the packet is verified by the client patch program, the patch process can start. If the data is compressed it first needs to be decompressed. In case of a full binary, the program can just be installed and run. If it is a patch, the running program needs to be shut down before it gets patched. When the patch process is done, the program can be started again.

An overview of the program structure can be seen in Figure 7.1.

7.6.1 Future improvements

- The server side needs to have a version table that keeps track of the versions of all the connected camera's software. It needs functionality that makes sure that a patch destinating to one camera does not go to other cameras. Instead of a server/client based approach, the patch program should be integrated into a package manager in order to be utilized by The Company.
- Another desirable function could be a check to see if the new or patched program still works after the patch process. Even though the patch succeeds, it does not mean that the new program will work as intended.
- A desirable mechanism considering the roll out could be to make the cameras cooperate with each other in their network just like dissemination in a

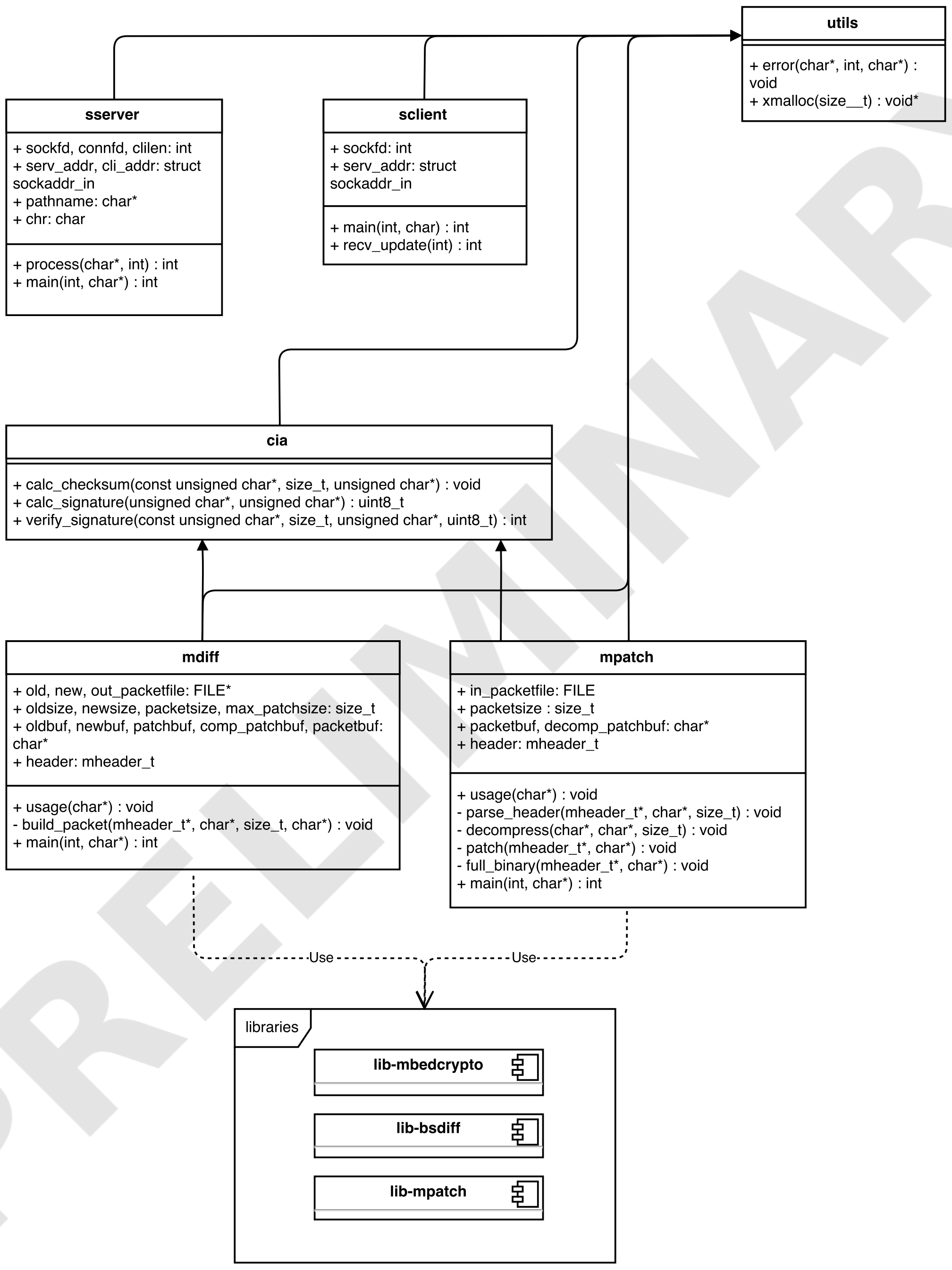

utils
+ error(char*, int, char*) : void
+ xmalloc(size__t) : void*
sserver
+ sockfd, connfd, clilen: int
+ serv_addr, cli_addr: struct sockaddr_in
+ pathname: char*
+ chr: char
+ process(char*, int) : int
+ main(int, char*) : int
sclient
+ sockfd: int
+ serv_addr: struct sockaddr_in
+ main(int, char) : int
+ recv_update(int) : int
cia
+ calc_checksum(const unsigned char*, size_t, unsigned char*) : void
+ calc_signature(unsigned char*, unsigned char*) : uint8_t
+ verify_signature(const unsigned char*, size_t, unsigned char*, uint8_t) : int
mdiff
+ old, new, out_packetfile: FILE*
+ oldsize, newsize, packetsize, max_patchsize: size_t
+ oldbuf, newbuf, patchbuf, comp_patchbuf, packetbuf: char*
+ header: mheader_t
+ usage(char*) : void
- build_packet(mheader_t*, char*, size_t, char*) : void
+ main(int, char*) : int
mpatch
+ in_packetfile: FILE
+ packetsize : size_t
+ packetbuf, decomp_patchbuf: char*
+ header: mheader_t
+ usage(char*) : void
- parse_header(mheader_t*, char*, size_t) : void
- decompress(char*, char*, size_t) : void
- patch(mheader_t*, char*) : void
- full_binary(mheader_t*, char*) : void
+ main(int, char*) : int
Use
Use
libraries
lib-mbedcrypto
lib-bsdiff
lib-mpatch

Figure 7.1: UML diagram of our patch program

Wireless Sensor Network. A group of cameras which all get the same update could announce a “master” that receives the update from the server and let the cameras cooperate to distribute the update to each other, thereby decresing the transmission rate from the server to the cameras with a factor of the number of cameras in the network. This means that the cameras must have knowledge about all other cameras in the network, as well as an implemented protocol for dissemination.

Chapter 8

Conclusion

Chapter 9

Crypto Benchmark

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
P3367	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

Table 9.1: The table shows benchmark results of different signing algorithms on different systems.

Bibliography

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

[2] C. J. Sreenan, S. Brown. “Software Updating in Wireless Sensor Networks: A Survey and Lacunae”. In: (2013).

[3] W. Dong, C. Chen, X. Liu, J. Bu. “Providing OS Support for Wireless Sensor Networks: Challenges and Approaches”. In: (2010).

[4] J. Paul Walters, Z. Liang, W. Shi, V. Chaudhary. “Wireless Sensor Network Security: A Survey”. In: (2006).

[5] 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).

[6] Jonathan W. Hui, David Culler. “The Dynamic Behavior of a Data Dissemination Protocolfor Network Programming at Scale”. In: (2004).

[7] 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`.

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

[9] 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).

[10] Times Higher Education. *Times Higher Education World University Rankings.* 2015. URL: `https://www.timeshighereducation.com/world-university-rankings/lund-university?ranking-dataset=133819` (visited on 12/01/2015).

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

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

[13] 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).

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

[15] G. Werner-Allen, K. Lorincz, M. Welsh, O. Marcillo, J. Johnson, M. Ruiz, J. Lees. "Deploying a Wireless Sensor Network on an Active Volcano". In: (2006).

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

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

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

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

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

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

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

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

[24] 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).

[25] 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).

[26] M. Bozorgi, L. K. Saul, S. Savage, and G. M. Voelker. "Beyond Heuristics: Learning to Classify Vulnerabilities and Predict Exploits". In: (2010).

[27] 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`.

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

[29] N. Gura, A. Patel, A. Wander, H. Eberle, S. Shantz. "Comparing elliptic curve cryptography and rsa on 8-bit cpus". In: (2004).

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

[31] 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`.

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

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

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

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

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

[37] QS WU. *QS Top Universities*. 2015. URL: `http://www.topuniversities.com/universities/lund-university` (visited on 12/01/2015).

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

Appendix A

Test Appendix

Some code here?