Software and Firmware Updates for Internet of Things

SIMON CARLSON

Master in Information Technology
Date: January 18, 2019
Supervisor: Amir Payberah
Examiner: Elena Dubrova
Swedish title: Detta är den svenska översättningen av titeln
School of Computer Science and Communication

Abstract

English abstract goes here.

Sammanfattning

Svensk sammanfattning här.

Contents

1 Introduction **1**
- 1.1 Background . . . 1
- 1.2 Problem Statement . . . 2
  - 1.2.1 Problem . . . 2
  - 1.2.2 Purpose . . . 3
  - 1.2.3 Goal . . . 3
- 1.3 Methodologies . . . 3
- 1.4 Risks, Consequences, and Ethics . . . 3
- 1.5 Scope . . . 4
- 1.6 Outline . . . 4

2 Theoretic Background **5**
- 2.1 IoT Network Stack . . . 5
  - 2.1.1 UDP . . . 6
  - 2.1.2 DTLS . . . 6
  - 2.1.3 CoAP . . . 8
- 2.2 Hardware/Firefly . . . 11
- 2.3 SUIT . . . 11
  - 2.3.1 Architecture . . . 11
  - 2.3.2 Information Model . . . 13
- 2.4 Contiki-NG . . . 17
  - 2.4.1 Processes, Events, And Memory Management . . . 18
  - 2.4.2 Timers . . . 19
  - 2.4.3 Networking Protocols in Contiki-NG . . . 19
- 2.5 EST CoAP . . . 20

3 Transportation of Firmware Images **21**

4 Updating of Firmware Images **22**

5 Evaluation and Results **23**

6 Discussion **24**

A Appendix Title **25**

Chapter 1

Introduction

1.1 Background

Internet of Things, or IoT, is the notion of connecting physical objects to the world-spanning Internet in order to facilitate services and communication both machine-to-human and machine-to-machine. By having everything connected, from everyday appliances to vehicles to critical parts of infrastructure, computing will be ubiquitous and the Internet of Things realized. The benefits from the IoT can range from quality of life services, such as controlling lights and thermostats from afar, to data gathering through wireless sensor networks, to enabling life critical operations such as monitoring a pacemaker. IoT is a broad definition and fits many different devices in many different environment, what they all have in common is that they are physical and connected devices.

In recent years the general public has become increasingly aware of digital attacks and intrustions affecting their day to day lives. In 2016 the DNS provider Dyn was attacked by a botnet consisting of heterogenous IoT devices infected by the Mirai malware. Devices such as printers and baby monitors were leveraged to launch an attack on Dyn's services which affected sites like Airbnb, Amazon, and CNN [15]. Cardiac devices implanted in patients were also discovered to be unsafe, with hackers being able to deplete the batteries of pacemakers prematurely [9]. These cases and many others make it clear that security in IoT is a big deal.

According to Bain & Company, the largest barrier for Internet of Things (IoT) adoptation is security concerns, and customers would

buy an average of 70% more IoT devices if they were secured [1]. Despite security being lacking today in IoT, the field is expected to grow rapidly and an increase in unsecured devices could spell disaster. Securing IoT equipment such as printers, baby monitors, and pacemakers is imperative to prevent future attacks. But what about the devices currently employed without security? They need to be updated and patched in order to fix these vulnerabilities, a non-trivial task.

1.2 Problem Statement

There is a need for secure software and firmware updates for IoT devices as vulnerabilities must be patched. For many IoT devices this mean patches must be applied over long distances and possibly unreliable communication channels as sending a technician to each and every device is unfeasible. There are non-open, proprietary solutions developed for specific devices but no open and interoperable standard. The IETF SUIT working group aims to define an architecture for such a mechanism without defining new transport or discovery mechanisms [20]. By expanding upon the work of the SUIT group, a standardized update mechanism suitable for battery powered, constrained, and remote IoT devices can be developed.

1.2.1 Problem

The thesis project will examine the architecture and information model proposed by SUIT in order to create and evaluate an updating mechanism for battery powered, constrained, and remote IoT devices with a life span exceeding five years. The update mechanism can be considered as two parts, communication and upgrading. Communication will happen over unreliable channels and the payloads must arrive intact and untampered with. The upgrade mechanism itself must ensure integrity of the target image and safely perform the upgrade with a limited amount of memory. The thesis will examine the update mechanism from the viewpoint of IoT devices running the Contiki-NG operating system on 32-bit ARM Cortex M3 processors. A suitable public key infrastructure developed by RISE will also be an underlying assumption for the work. The thesis aims to investigate the problem "How can the SUIT architecture be used to provide secure software and firmware update for the Internet of Things?".

1.2.2 Purpose

The purpose of the thesis project is to provide an open and interoperable software and firmware update mechanism that complies with the standards suggested by the IETF SUIT working group. This will aid other projects trying to secure their IoT devices following accepted standards.

1.2.3 Goal

The goals of the thesis are to study current solutions and then propose lightweight end-to-end protocols that can be used when updating IoT devices. The degree project shall deliver specifications and prototypes of the protocols in an IoT testbed, as well as a thesis report.

1.3 Methodologies

The thesis will follow a mix of qualitative and quantitative methodologies. The SUIT group defines some goals or constraints a suitable updating mechanism should follow, but as their proposed architecture is agnostic of any particular technology these goals cannot be easily quantified. It is better to regard them as qualitative properties the mechanism should have. In addition to this there are some relevant measurements, such as reliability of the communication and memory and power requirements of the updates, that can be used in an evaluation. The quantitative part of the evaluation will follow an objective, experimental approach, while the qualitative part will follow a interpretative approach.

1.4 Risks, Consequences, and Ethics

IoT devices need to be securely updated in order to fix vulnerabilities and undergo maintenance to ensure their future operability. Failing to do so could leave a device or network unsecured, posing a security risk. Furthermore the performance of the service provided by the IoT device may itself degrade. Lastly, if updates are not applied in a safe and secure manner, the updating mechanism itself might cause service disruptions. These risks are serious and could pose ethical dilemmas depending on what the IoT device is used for.

1.5 Scope

The updating mechanism can be roughlt split into three parts: the actual updating or flipping of images on the device, the transportation of the new image, and managing a heterogenous network of devices needing different updates at different times. This thesis will only look at the first two parts, updating a firmware image on the device and transportation of the image. The thesis will not be concerned about device management.

1.6 Outline

Chapter two describes the theoretical background needed to understand the results of the thesis. This includes the network protocols, hardware, operating system, and PKI being used in the thesis. Chapter three describes the design of the communication protocols used in the update mechanism. Chapter four describes how the local update mechanism works and how devices can upgrade their images. Chapter five evaluates the developed updating mechanism and presents the results. Chapter six ends the report with a discussion about the update mechanism and its results.

Chapter 2

Theoretic Background

This chapter presents the theoretical background needed to understand the thesis. Section 2.1 introduces the networks protocols used and motivates their use over other protocols in an IoT context. Section 2.2 presents the target hardware for this thesis. The following section, Section 2.3 presents and explains the SUIT architecture and information model and their respective requirements as formulated by the IETF. Section 2.4 presents the Contiki-NG operating system which the updating mechanism will be developed for, and finally 2.5 introduces EST-CoAP which is a protocol for public key certificate distribution.

2.1 IoT Network Stack

Network protocols in IoT networks operate under different circumstances compared to traditional computer networks. Networks in IoT are defined by their low power requirements, low reliability, and low computational performance on edge devices. This posts some constraints on the protocols used as they must be suitable for use in IoT networks. One of the most widely used network protocol stacks today in traditional networks is the TCP/IP stack. It uses TCP as a transportation protocol, usually with TLS for security, and a common application layer protocol is HTTP. TCP is however poorly suited for IoT networks as it is a connection based, stateful protocol which tries to ensure the guaranteed delivery of packets in the correct order. There is also advanced congestion control mechanisms in TCP which are hard to apply on low-bandwitch, unreliable networks.

IoT networks on the other hand often utilize UDP for transport.

UDP is also an IP-based protocol which grants some interoperability with traditional networks while performing better in IoT contexts. As TLS is based on the same assumptions TCP does it is unsuitable for UDP networks. Instead there is DTLS, which is a version of TLS enhanced for use in datagram oriented protocols such as UDP. HTTP can be used over UDP for the application layer, but as HTTP is encoded in human readable plaintext it is unecessarily wordy and not optimal for constrained networks. Instead, CoAP is a common protocol for the application layer in IoT networks. Subsection 2.1.1 explains UDP and why UDP is preferable in IoT networks. Subsection 2.1.2 briefly explains TLS, why it is unsuitable for IoT networks, the differences between TLS and DTLS and why DTLS is used. Lastly subsection 2.1.3 describes the CoAP protocol.

2.1.1 UDP

UDP is a stateless and asynchronous transfer protocol for IP [16]. It does not provide any reliability mechanisms but is instead a best-effort protocol. It also does not guarantee delivery of messages. For general purposes in unconstrained environments TCP is usually the favored transport protocol as it is robust and reliable, but in environments where resources are scarce and networks unreliable, a stateful protocol like TCP could face issues. Since TCP wants to ensure packet delivery, it will retransmit packages generating a lot of traffic and processing required for a receiver. Also, if the connection is too unstable TCP will not work at all since it can no longer guarantee the packets arrive. The best-effort approach of UDP is favorable in these situations, in addition to UDP being a lightweight protocol.

Figure 2.1 shows the UDP header format. The source port is optional, the length denotes the length of the datagram (including the header), and the checksum is calculated on a pseudo-header constructed from both the IP header, UDP header, and data.

2.1.2 DTLS

DTLS is a protocol which adds privacy to datagram protocols like UDP [18]. The protocol is designed to prevent eavesdropping, tampering, or message forgery. DTLS is based on TLS, a similar protocol for stateful transport protocols such as TCP, which would not work well on unre-

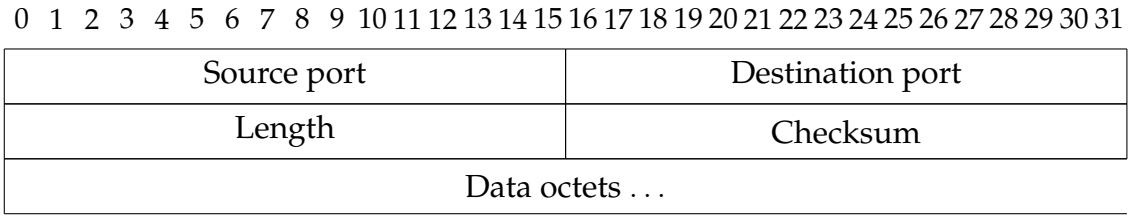

Figure 2.1: The UDP header format.

liable networks. The main issues with using TLS over unreliable networks is that TLS decryption is dependant on previous packets, meaning decryption of a packet would fail if the previous packet was not received, as well as the TLS handshake procedure assumes all handshake messages are delivered reliably.

DTLS solves this by banning stream ciphers, effectively making decryption an independent operation between packets, as well as adding explicit sequence numbers. Furthermore, DTLS supports packet retransmission, reordering, as well as fragmenting DTLS handshake messages into several DTLS records. These mechanisms makes the handshake process feasible over unreliable networks. By splitting messages into different DTLS records, fragmentation at the IP level can be avoided since a DTLS record is guaranteed to fit an IP datagram.

Listing 2.1 shows the DTLS record structure. It contains a TLS 1.2 type field, a version field which for DTLS 1.2 is 254.253, an epoch counter that is incremented for each cipher state change, an increasing sequence number (unique to each epoch), a length field and a fragment field containing the application data [17]. These fields, with the exception of the epoch and sequence number, are the same as in TLS 1.2 [18].

Listing 2.1: The DTLS plaintext record structure.

```
struct {
    ContentType type;
    ProtocolVersion version;
    uint16 epoch;
    uint48 sequence_number;
    uint16 length;
    opaque fragment[DTLSPlaintext.length];
} DTLSPlaintext;
```

Listing 2.2: The DTLS ciphertext record structure.

```
struct {
    ContentType type;
    ProtocolVersion version;
    uint16 epoch;
    uint48 sequence_number;
    uint16 length;
    select (CipherSpec.cipher_type) {
        case block: GenericBlockCipher;
        case aead: GenericAEADCipher;
    } fragment;
} DTLSCiphertext;
```

TLS records are compressed and then encrypted, the same holds for DTLS. When initating contact, a compression algorithm is chosen. The DTLSPlaintext is then compressed into a DTLSCompressed record, with similar structure to Listing 2.1.The compressed record is encrypted into a DTLSCiphertext record whose structure is shown in Listing 2.2. Note that since DTLS disallows stream ciphers they are not an option in the encrypted fragment, whereas in TLS they are.

In order to communicate via TLS and DTLS, a handshake has to be carried out. The handshake establishes parameters such as protocol version, cryptographic algorithms, and shared secrets. The TLS handshake involves hello messages for establishing algorithms, exchanging random values, and checking for earlier sessions. Then cryptographic parameters are shared in order to agree on a shared premaster secret. The parties authenticate each other via public key encryption, generate a shared master secret based on the premaster secret, and finally verifies that their peer has the correct security parameters. The DTLS handshake adds to this a stateless cookie exchange to prevent DoS attacks, some modifications to the handshake header to make communication over UDP possible, and retransmission timers since the communication is unreliable. Otherwise the DTLS handshake is as the TLS handshake.

2.1.3 CoAP

CoAP is an application layer protocol designed to be used by constrained devices over networks with low throughput and possibly high unreliability for machine-to-machine communication [19]. While de-

signed for constrained networks, a design feature of CoAP is how it is easily interfaced with HTTP so that communication over traditional networks can be proxied. Furthermore, CoAP is a REST based protocol utilizing application endpoints, a subset of standardised request/response codes, URIs, and MIME types [8]. Additionally CoAP offers features such as multicast support, asynchronous messages, low header overhead, and UDP and DTLS bindings which are all suitable for constrained environments.

As CoAP is usually implemented on top of UDP, communication is stateless and asynchronous. For this reason CoAP defines four message types: Confirmable, Non-confirmable, Acknowledgement, and Reset. These message types are combined with a subset of HTTP method codes. Confirmable messages must be answered with a corresponding Acknowledgement, this provides one form of reliability over an otherwise unreliable channel. Non-confirmable messages do not require an Acknowledgement and thus act asynchronously. Reset messages are used when a recipient is unable to process a Non-confirmable message. Confirmable, Non-confirmable, and Acknowledgement messages all use Message IDs in order to detect duplicate messages in case of retransmission.

Since CoAP is based on unreliable means of transport, there are some lightweight reliability and congestion control mechanics in CoAP. Message IDs allows for detection of duplicate messages and tokens allow asynchronous requests and responses be paired correctly. There is also a retransmission mechanic with an exponential back-off timer for Confirmable messages so that lost Acknowledgements does not cause a flood of retransmissions.

A feature of CoAPs messaging model is the piggybacked respones. If a response to a Confirmable or Non-confirmable request is immediately available and fits in the Acknowledgement, the response itself can be delivered with the Acknowledgement. If the response is not available, a recepient can respond with an empty Acknowledgement and later send a Confirmable message containing the response. Requests use the GET, PUT, POST, and DELETE methods in a manner that is very similar but not identical to HTTP.

Figure 2.2 shows the CoAP message format. The 2-bit version (Ver) field indicates the CoAP version, which at time of writing is 1 (01 in binary). The 2-bit type (T) field determines the type of message (Confirmable, Non-confirmable, Acknowledgement, Reset). Token length

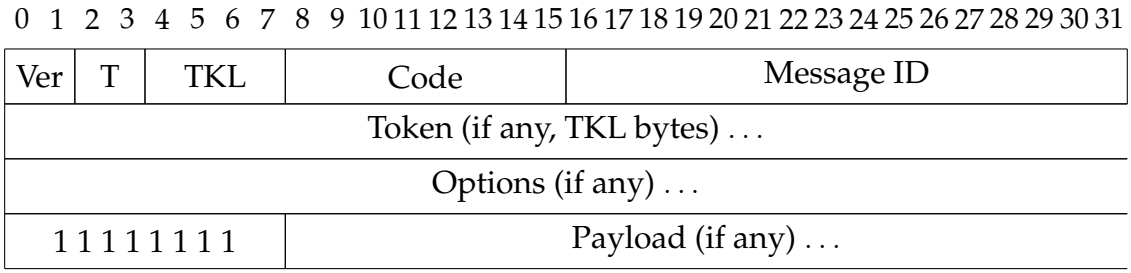

Figure 2.2: The CoAP message format.

(TKL) indicates the length of the Token field which can vary between zero to eight bytes. The 8-bit code field carries which method code the message carries and can be further broken into a 3-bit class and 5-bit detail. The class can indicate a request (0), a success response (2), a client error response (4), or a server error response (5), with the detail further specifying the status of the message. The message ID is a 16-bit integer used to detect duplicate message and to match Acknowledgement or Resets with their initiating requests.

Following the header is the zero to eight bit Token value, which in turn is follow by zero or more Options. Lastly comes the optional payload, which if present is prefixed by a payload marker (0xFF). The length of the payload is dependant on the carrying protocol, which in this thesis will be DTLS. The length of the payload is calculated depending on the size of the CoAP header, token, and options as well as maximum DTLS record size.

Since firmware images can be relatively large their size needs to be handled during transportation. UDP and DTLS only supports fragmentation which can be problematic in unreliable networks. To remedy this CoAP supports block-wise transfers [3]. A Block option allows stateless transfer of a large file separated in different blocks. Each block can be individually retransmitted and by using monotonically increasing block numbers, the blocks can be reassembled. The size of blocks can also be negotiated between server and client meaning they can always find a suitable block size, making the mechanism quite flexible.

2.2 Hardware/Firefly

2.3 SUIT

The IETF SUIT (Software Updates for Internet of Things) working group aims to define a firmware update solution that is interoperable and non-proprietary [20]. The solution shall be usable on Class 1 devices as defined in RFC 7228. These devices feature 10 KiB of RAM and 100 KiB code size, which makes it suitable for this thesis [2]. The solution may be applied to more powerful devices. The working group does not however try to define new transport or discovery mechanism, making their proposal angostic of any particular technology. Subsection 2.3.1 presents the SUIT architecture and subsection 2.3.2 presents the SUIT information model.

2.3.1 Architecture

There is an Internet Draft by the SUIT group focusing on the firmware update architecture [14]. This draft describes the goals and requirements of the architecture, although makes no mention of any particular technology. The overarching goals of the update process is to thwart any attempts to flash unauthorized, possibly malicious firmware images as well as protecting the firmware image's confidentiality. These goals reduces the possibility of an attacker either getting control over a device or reverse engineering a malicious but valid firmware image as an attempt to mount an attack.

In order to accept an image and update itself, a device must be presented with certain information. Several decisions must be made before updating and the information comes in form of a manifest. The next section will describe the requirements posted upon this manifest in more detail. The manifest helps the device make important decisions such as if it trusts the author of the new image, if the image is intact, if the image is applicable, where the image should be stored and so on. This in turns means the device also has to trust the manifest itself, and that both manifest and update image must be distributed in a safe and trusted architecture. The draft [14] presents ten qualitative requirements this architecture should have:

- Agnostic to how firmware images are distributed:

As this thesis aims to implement a prototype of an update mechanism, some choices about technology has to be done. This will realistically mean only a subset of the SUIT standard will be implemented as certain parts of the standard is not applicable. The proposed network stack uses UDP, DTLS, and CoAP for transportation and the target devices are Firefly devices running the Contiki-NG operating system.

- Friendly to broadcast delivery:
  Broadcasting will not be of main concern in this thesis.
- Use state-of-the-art security mechanisms:
  The SUIT standard assumes a PKI is in place. RISE has previously developed a PKI suitable for IoT, this PKI is an underlying assumption for the thesis. The PKI will allow for signing of the update manifest and firmware image.
- Rollback attacks must be prevented:
  The manifest will contain metadata such as monotonically increasing sequence numbers and best-before timestamps to avoid rollback attacks.
- High reliability:
  This is an implementation requirement and depends heavily on the hardware of the target device.
- Operate with a small bootloader:
  This is also an implementation requirement.
- Small parser:
  It must be easy to parse the fields of the update manifest as large parser can get quite complex. Validation of the manifest will happen on the constrained devices which further motivates a small parser and thus less complex manifests.
- Minimal impact on existing firmware formats:
  The update mechanism itself must not make assumptions of the current format of firmware images, but be able to support different types of firmware images.
- Robust permissions:
  This requirement is directed towards the administration of firmware

updates and how different roles interact with the devices. The thesis will not consider any infrastructure outside of transporting manifest and image and applying the update such as device management, but will consider authorisation of parties through techniques like signing.

- Operating modes:
  The draft presents three broad modes of updates: client-initiated updates, server-initiated updates, and hybrid updates, where hybrids are mechanisms that require interaction between the device and firmware provider before updating. The thesis will look into all three of these broad classes.

An example architecture encompassing a device, a firmware update author, a firmware server, a network operator and a device operator is presented in the draft. The author and the device ineract with the firmware server in order to communicate firmware updates and possibly manifests. The communication of the device as well as the updating itself is the concern of this thesis. The draft also presents device management for the device operator so that it is possible to track the state of updates in a network. Device management is considered out of scope for this thesis.

The distribution of manifest and firmware image is also discussed, with a couple of options being possible. The manifest and image can be distributed together to a firmware server. The device then recieves the manifest either via pulling or pushing and can subsequently download the image. Alternatively, the manifest itself can be directly sent to the device without a need of a firmware server, while the firmware image is put on the firmware server. After the device has receieved the lone manifest through some method, the firmware can be downloaded from the firmware server. The SUIT architecture does not enforce a specific method to be used when delivering the manifest and firmware, but states that an update mechanism must support both types.

2.3.2 Information Model

The corresponding Internet Draft for the information model presents the information needed in the manifest to secure a firmware update mechanism [13]. It also presents threats, classifies them according to

the STRIDE model, and presents security requirements that map to the threats [12]. Furthermore it presents use cases and maps usability requirements to the use cases. Finally it presents the proposed elements of the manifest. Since the thesis makes a choice about specific technologies to use, not all use cases, usability requirements, and manifest elements are deemed necessary. The threats and security requirements however are. Note that the information model does not discuss threats outside of transporting the updates, such as physical attacks.

Table 2.1 shows how the security threats map to security requirements along with a brief description of each threat and their class. All threats, MFT1 - MFT12, are considered relevant for this project and are what the SUIT information model attempts to prevent. The security requirements, with brief descriptions, and how they map to the manifest elements can be found in Table 2.2. Finally, Table 2.3 shows the proposed manifest elements, their status, and which security requirements each element implements.

Table 2.1: Mapping security threats to security requirements.

Threat	Description	Threat Model	Mitigated By
MFT1	Old Firmware	Elevation of Privilege	MFSR1
MFT2	Mismatched Firmware	Denial of Service	MFSR2
MFT3	Offline device + old firmware	Elevation of Privilege	MFSR3
MFT4	The target device misinterprets the type of payload	Denial of Service	MFSRa
MFT5	The target device installs the payload to the wrong location	Denial of Service	MFSR4b
MFT6	Redirection	Denial of Service	MFSR4c
MFT7	Payload Verification on Boot	Elevation of Privilege	MFSR4d
MFT8	Unauthenticated Updates	Elevation of Privilege	MFSR5
MFT9	Unexpected Precursor Image	Denial of Service	MFSR4e

Table 2.1: Mapping security threats to security requirements.

Threat	Description	Threat Model	Mitigated By
MFT10	Unqualified Firmware	Denial of Service, Elevation of Privilege	MFSR6, MFSR8
MFT11	Reverse Engineering Of Firmware Image for Vulnerability Analysis	All classes	MFSR7
MFT12	Overriding Critical Manifest Elements	Elevation of Privilege	MFSR8

Table 2.2: Mapping security requirements to manifest elements.

Security Requirement	Description	Implemented By	Mitigates
MFSR1	Monotonic Sequence Numbers	Monotonic Sequence Number	MFT1
MFSR2	Vendor and Device-type Identifiers	Vendor ID Condition, Class ID Condition	MFT2
MFSR3	Best-Before Timestamps	Best-Before timestamp condition	MFT3
MFSR4a	Authenticated Payload Type	Payload Format, Storage Location	MFT4
MFSR4b	Authenticated Storage Location	Storage Location	MFT5
MFSR4c	Authenticated Remote Resource Location	URIs	MFT6
MFSR4d	Secure Boot	Payload Digest, Size	MFT7
MFSR4e	Authenticated precursor images	Precursor Image Digest Condition	MFT9
MFSR4f	Authenticated Vendor and Class IDs	Vendor ID Condition, Class ID Condition	MFT2

Table 2.2: Mapping security requirements to manifest elements.

Security Requirement	Description	Implemented By	Mitigates
MFSR5	Cryptographic Authenticity	Signature, Payload Digest	MFT8
MFSR6	Rights Require Authenticity	Signature	MFT10
MFSR7	Firmware Encryption	Content Key Distribution Method	MFT11
MFSR8	Access Control Lists	Client-side code (not specified in manifest)	MFT10, MFT12

Table 2.3: Manifest elements, their status, and which security requirements they implement.

Manifest Element	Status	Implements Requirements
Version identifier	MANDATORY	
Monotonic Sequence Number	MANDATORY	MFSR1
Precursor Image Digest Condition	MANDATORY (for differential updates)	MFSR4e
Payload Format	MANDATORY	MFSR4a, MFUR5
Storage Location	MANDATORY	MFSR4b
Payload Digest	MANDATORY	MFSR4d, MFUR8
Size	MANDATORY	MFSR4d
Signature	MANDATORY	MFSR5, MFSR6, MFUR6
Dependencies	MANDATORY	MFUR3
Content Key Distribution Method	MANDATORY (for encrypted payloads)	MFSR7
Vendor ID Condition	OPTIONAL	MFSR2, MFSR4f

Table 2.3: Manifest elements, their status, and which security requirements they implement.

Manifest Element	Status	Implements Requirements
Class ID Condition	OPTIONAL	MFSR2, MFSR4f
Required Image Version List	OPTIONAL	MFUR7
Best-Before Timestamp Condition	OPTIONAL	MFSR3
Component Identifier	OPTIONAL	MFUR3
URIs	OPTIONAL	MFSR4c
Directives	OPTIONAL	MFUR1
Aliases	OPTIONAL	MFUR2
Processing Steps		MFUR6
XIP Address		MFUR8

To summarize, the SUIT information model proposes to use a signed manifest that is distributed to each device in need of an update. The device then processes the manifest in order to determine if the update is trusted, suitable, up to date, with many other optional elements such as if other precursor images, special processing steps, or new URIs to fetch the images are needed. The model does not make assumptions about technology which is one of the reasons there are optional elements, not all of them are applicable to all solutions. Nevertheless, the architecture and information model together provides a solid base on which to design a secure update mechanism for IoT.

2.4 Contiki-NG

Contiki-NG is an open-source operating system for resource constrained IoT devices based on the Contiki operating system [4], [6]. Contiki-NG focuses on low-power communication and standard protocols and comes with IPv6/LoWPAN, DTLS, and CoAP implementations which makes it a suitable operating system for this thesis. Furthermore, Contiki-NG is open source and licensed under the permissive BSD 3-Clause license and targets a wide variety of boards which makes it align with

SUITs goal of creating an open standard for updating IoT devices.

Subsection 2.4.1 explains processes, events, and memory management in Contiki-NG. These internals are heavily based on the previous Contiki operating system and much of the information is gathered from there. Subsection 2.4.2 presents the different timers available in Contiki-NG and their different usages. Finally, subsection 2.4.3 presents part of the network protocol stack that is implemented in standard Contiki-NG.

2.4.1 Processes, Events, And Memory Management

Contiki-NG has a process abstraction which is built on lightweight protothreads [7]. A process is declared through a `PROCESS` macro and can be automatically started after system boot or when a specific module is loaded. User-space processes are run in a cooperative manner while kernel-space processes can preempt user-space processes. Contiki-NGs execution model is event based, meaning processes often yield execution until they are informed a certain event has taken place, upon which they can act. Examples of events are timers expiring, a process being polled, or a network packet arriving.

Contiki-NG provides to memory allocators in addition to using static memory. They are called `MEMB` and `HeapMem` and are semi-dynamic and dynamic, respectively. `MEMB` provides ways to manage memory blocks. The memory blocks are allocated on static memory as arrays of constant sized objects. After a memory block has been declared, it is initialized after which objects can be allocated memory from the block. All objects allocated through the same block have the same size. Blocks can be freed, and it is possible to check whether a pointer resides within a certain memory block or not.

`HeapMem` solves the issue of dynamically allocating objects of varying sizes during runtime in Contiki-NG. It can be used on a variety of hardware platforms, something a standard C `malloc` implementation could struggle with. To allocate memory on the heap, the number of bytes to allocate must be provided and a pointer to a contiguous piece of memory is returned (if there is enough contiguous memory). Memory can be reallocated and deallocated such as using a normal `malloc`.

2.4.2 Timers

Contiki-NG provides different timers used by both the kernel and user-space applications. The timers are based on the clock module which is responsible for handling system time. The definition of system time is platform dependent.

The different timers and their usages are:

- `timer`: a timer without built-in notification.
- `stimer`: counts in seconds and has a long wrapping period.
- `etimer`: schedules events to processes. Since events in Contiki-NG can put yielding processes in the execution state, `etimer` can be used to periodically schedule code by setting up an `etimer` to signal a specific event when it expires then waiting on that event
- `ctimer`: schedules calls to a callback function. `ctimer` works in similar ways to etimer but with callbacks to callback functions instead of events.
- `rtimer`: schedules real-time tasks. Since real-time tasks face different requirements than normal tasks, rtimer uses a higher resolution clock. Real-time tasks preempt normal execution so that the real-time task can execute immediately. This mean there are constraints on what can be done in real-time tasks as many functions cannot handle preemption.

`timer`, `stimer`, and `rtimer` as stated to be safe from interrupts while `etimer` and `ctimer` are unsafe. All timers are declared using a timer struct, which is also how the timer is interacted with.

2.4.3 Networking Protocols in Contiki-NG

Contiki-NG features a IPv6 network stack designed for unreliable, low-power IoT networks. There are many protocols implemented in the stack, this thesis will look at UDP and CoAP secured by DTLS. Beneath IPv6 Contiki-NG supports IEEE 802.15.4 with TSCH [10].

The CoAP implementation in Contiki-NG is based on Erbium by Mattias Kovatsch but has become part of core modules in the operating system itself [11]. The default implementation supports both unsecured (CoAP) and secured (CoAPs) communication. CoAPs uses a

DTLS implementation called TinyDTLS which handles encryption and decryption of messages [5]. The CoAP implementation consists of:

- a CoAP engine which registers CoAP resources
- a CoAP handler API that allows for implementations of resource handlers. The handlers act upon incoming messages to their corresponding resource
- a CoAP endpoint API allowing handling of different kinds of CoAP endpoints.
- a CoAP transport API which hands CoAP data from the CoAP stack to the transport protocol.
- CoAP messages functions for parsing and creating messages
- a CoAP timer API providing timers for retransmission mechanisms.

2.5 EST CoAP

Chapter 3

Transportation of Firmware Images

Chapter 4

Updating of Firmware Images

Chapter 5

Evaluation and Results

Chapter 6

Discussion

Bibliography

[1] Syed Ali, Ann Bosche, and Frank Ford. *Cybersecurity Is the Key to Unlocking Demand in the Internet of Things*. Oct. 2018. URL: https://www.bain.com/insights/cybersecurity-is-the-key-to-unlocking-demand-in-the-internet-of-things/ (visited on 12/10/2018).

[2] Carsten Bormann, Mehmet Ersue, and Ari Keränen. *Terminology for Constrained-Node Networks*. RFC 7228. May 2014. DOI: 10.17487/RFC7228. URL: https://rfc-editor.org/rfc/rfc7228.txt.

[3] Carsten Bormann and Zach Shelby. *Block-Wise Transfers in the Constrained Application Protocol (CoAP)*. RFC 7959. Aug. 2016. DOI: 10.17487/RFC7959. URL: https://rfc-editor.org/rfc/rfc7959.txt.

[4] contiki-ng. *Contiki-NG*. https://github.com/contiki-ng/contiki-ng. 2019. (Visited on 01/18/2019).

[5] contiki-ng. *TinyDTLS*. https://github.com/contiki-ng/tinydtls. 2018. (Visited on 01/18/2019).

[6] contiki-os. *Contiki*. https://github.com/contiki-os/contiki. 2018. (Visited on 01/18/2019).

[7] Adam Dunkels et al. "Protothreads: Simplifying Event-driven Programming of Memory-constrained Embedded Systems". In: *Proceedings of the 4th International Conference on Embedded Networked Sensor Systems*. SenSys '06. Boulder, Colorado, USA: ACM, 2006, pp. 29–42. ISBN: 1-59593-343-3. DOI: 10.1145/1182807.1182811. URL: http://doi.acm.org/10.1145/1182807.1182811.

[8] Roy T Fielding and Richard N Taylor. *Architectural styles and the design of network-based software architectures*. Vol. 7. University of California, Irvine Irvine, USA, 2000.

[9] Alex Hern. *Hacking risk leads to recall of 500,000 pacemakers due to patient death fears*. Aug. 2017. URL: https://www.theguardian.com/technology/2017/aug/31/hacking-risk-recall-pacemakers-patient-death-fears-fda-firmware-update (visited on 01/16/2019).

[10] "IEEE Standard for Information Technology - Telecommunications and Information Exchange Between Systems - Local and Metropolitan Area Networks Specific Requirements Part 15.4: Wireless Medium Access Control (MAC) and Physical Layer (PHY) Specifications for Low-Rate Wireless Personal Area Networks (LR-WPANs)". In: *IEEE Std 802.15.4-2003* (2003). DOI: 10.1109/IEEESTD.2003.94389.

[11] M. Kovatsch, S. Duquennoy, and A. Dunkels. "A Low-Power CoAP for Contiki". In: *2011 IEEE Eighth International Conference on Mobile Ad-Hoc and Sensor Systems*. Oct. 2011, pp. 855–860. DOI: 10.1109/MASS.2011.100.

[12] Microsoft. *The STRIDE Threat Model*. Nov. 2009. URL: https://docs.microsoft.com/en-us/previous-versions/commerce-server/ee823878(v=cs.20) (visited on 01/17/2019).

[13] Brendan Moran, Hannes Tschofenig, and Henk Birkholz. *Firmware Updates for Internet of Things Devices - An Information Model for Manifests*. Internet-Draft draft-ietf-suit-information-model-01. Work in Progress. Internet Engineering Task Force, July 2018. 29 pp. URL: https://datatracker.ietf.org/doc/html/draft-ietf-suit-information-model-01.

[14] Brendan Moran et al. *A Firmware Update Architecture for Internet of Things Devices*. Internet-Draft draft-ietf-suit-architecture-02. Work in Progress. Internet Engineering Task Force, Jan. 2019. 22 pp. URL: https://datatracker.ietf.org/doc/html/draft-ietf-suit-architecture-02.

[15] Nicole Perlroth. *Hackers Used New Weapons to Disrupt Major Websites Across U.S.* Oct. 2016. URL: https://www.nytimes.com/2016/10/22/business/internet-problems-attack.html (visited on 01/16/2019).

[16] Jon Postel. *User Datagram Protocol*. RFC 768. Aug. 1980. DOI: 10.17487/RFC0768. URL: https://rfc-editor.org/rfc/rfc768.txt.

[17] Eric Rescorla and Tim Dierks. *The Transport Layer Security (TLS) Protocol Version 1.2*. RFC 5246. Aug. 2008. DOI: 10.17487/RFC5246. URL: https://rfc-editor.org/rfc/rfc5246.txt.

[18] Eric Rescorla and Nagendra Modadugu. *Datagram Transport Layer Security Version 1.2*. RFC 6347. Jan. 2012. DOI: 10.17487/RFC6347. URL: https://rfc-editor.org/rfc/rfc6347.txt.

[19] Zach Shelby, Klaus Hartke, and Carsten Bormann. *The Constrained Application Protocol (CoAP)*. RFC 7252. June 2014. DOI: 10.17487/RFC7252. URL: https://rfc-editor.org/rfc/rfc7252.txt.

[20] *Software Updates for Internet of Things (suit)*. URL: https://datatracker.ietf.org/wg/suit/about/ (visited on 01/16/2019).

Appendix A

Appendix Title