Master's Thesis

Authorization for Industrial Control Systems

Niklas Hjern
Jonas Vistrand

Department of Electrical and Information Technology,
Faculty of Engineering, LTH, Lund University, September 2014.

Authorization for Industrial Control Systems

Niklas Hjern
hjern.niklas@gmail.com
Jonas Vistrand
ael09jvi@student.lu.se

Department of Electrical and Information Technology
Lund University

Advisor:
Martin Hell
EIT
Ludwig Seitz
SICS
Andreas Bäckman
ABB

September 24, 2014

Printed in Sweden
E-huset, Lund, 2014

Abstract

Every day more and more devices are getting connected to the Internet, a phenomenon commonly referred to as the Internet of Things[1]. Since security and privacy are more important than ever before this presents an interesting problem. Suddenly devices with not even near as much computing power as a desktop computer are tasked with performing heavy security computations designed to be used in powerful systems with little resource and power limitations. This thesis presents a solution for performing authorization for a resource limited system using a trusted third party, thus transferring the usually quite heavy authorization computations from a resource constrained device to another device where no such restrictions exists. When a client wishes to request a resource on the constrained device it must first retrieve authorization information from the third party and include this in the request. The authorization information is then validated by confirming that it originates from the trusted third party using a shared secret. In this thesis the constrained system is represented by an ABB control system of model 800xA and by transferring the authorization cost to another system the increased amount of resource usage on this device is kept to a minimum. It is also shown that this increase is negligible compared to the increase in resource usage when authentication and message protection in the form of TLS was implemented.

Sammanfattning

Varje dag kopplas fler och fler apparater upp till internet, ett fenomen som vanligtvis kallas för Sakernas Internet[1]. Eftersom att säkerhet och integritet är viktigare än det någonsin varit förut bidrar detta till ett intressant problem. Plötsligt måste apparater med långt ifrån lika mycket processorkraft som en dator utföra tunga beräkningar som är designade att användas i kraftfulla system med små resurs- och kraftbegränsningar. Detta examensarbete lägger fram en teori om hur man kan implementera ett åtkomstsystem på ett resursbegränsat system genom att använda en betrodd tredje part och således överföra de vanligtvis tunga beräkningarna från det resursbegränsade systemet till ett annat system utan sådana begränsningar. När en klient vill begära åtkomst till ett begänsat system måste den först hämta behörighetsinformation från den tredje parten och inkludera denna information i begäran. Behörighetsinformationen valideras sedan genom att verifiera att resultatet härstammar från den tredje parten med hjälp av en delad hemlighet. I det här examensarbetet representeras det resursbegränsade systemet av ABBs kontrollsystem 800xA och genom att överföra kostnaden för åtomstberäkningarna till ett annat system hålls den ökade resursförbrukningen till ett minimum. Det visas också att denna ökning är försumbar jämfört med ökningen av resursförbrukning när autentisering och meddelandeskydd i form av TLS implementerades.

Contents

List of Figures

List of Tables

Chapter 1
Introduction

In a world where the Internet usage is growing at a rate never before seen along with more and more devices getting connected to the Internet, the need for security and privacy has never been greater. This relatively new occurrence of connecting devices other than computers to the Internet, like e.g. cellphones, alarms and game consoles, is commonly known as the Internet of Things [1]. Connecting these devices to the Internet provides a challenge not only because that they are much more resource limited than the standard computer but also because of the number of devices connected to networks will greatly increase, increasing the load on said networks, leading to further constraints. Despite their resource constraints many of these devices are perfectly capable of handling security measures, but doing so will impede the devices regular functionality making implementing them infeasible. Therefore any costly security measures on these devices must be streamlined or kept to a minimum, security measures that in many cases are an absolute necessity. Imagine having your pacemaker, a very resource constrained device, connected to a network. It would then be of utmost importance to implement some sort of authorization scheme on the device, deciding who should gain access to the device and what resources users should have access to. An authorization scheme for the previously mentioned example could consist of giving your doctor access to the device's read and write resources, you as a user access to the read resource and deny access to all the resources for everyone else. This means that your doctor would both be able to read your heart-rate data and set the pacing mode of your pacemaker, whereas you would only be able to read your heart-rate data.

1.1 Purpose and Goals

The purpose of this project is to construct and implement an authorization system with secure communication on a system with constrained resources, such as RAM, flash memory and network capacity. The access control mechanism is based on a design suggested in [2] and is to be implemented on a control system provided by ABB. Traditionally when a user tries to access a protected resource on a remote system, the access control evaluation is done on the protected system itself, an evaluation that can be quite costly in terms of computing time. In many cases the remote system will be a computer without any real resource limitations, thus performing such an evaluation would not present a problem. However in the world we

live in now, not only computers are connected to the Internet but also embedded systems designed to perform specific tasks and to be power efficient while doing them, making resource limitations an impending problem. In [2] it is therefore suggested that the costly access control evaluation is to be done on a trusted third party, thus transferring this cost onto a system with no resource limitations.

The theory tested in this project is that this authorization system is much more cost efficient for a resource limited system to perform than doing a traditional access control policy evaluation. Hopefully this cost can be shown to be negligible compared to the cost of authenticating a client, which is a prerequisite for an authorization system, and encrypting the communication, preventing man-in-the-middle attacks. The cost of performing the evaluation will be measured in time, heap utilization and program size on the control system provided by ABB.

1.2 About SICS

This Master Thesis was conducted in collaboration with the Swedish Institute of Computer Science (SICS). SICS is a non-profit research institute for applied information and communication technology. By conducting research in the aforementioned fields, in collaboration with small and large companies, their goal is to strengthen the competitiveness of industrial companies located in Sweden. These goals are to be achieved by suggesting, developing and promoting new research technologies, both towards the industry but also towards society at large.[3]

1.3 About ABB

This Master Thesis was also conducted in collaboration with ABB, one of the largest power and automation companies in the world with approximately 150,000 employees worldwide. ABB's activity can be divided into five different divisions that all cater to specific industries. ABB has a long history of inventing and creating new technologies that now form the basis of our modern society. As of now they are the worlds largest supplier of e.g. industrial motors and drives, generators to the wind industry and power grids worldwide.[4]

1.4 Report Structure

Chapter 2 - Contains the background theory of several different technologies used in this thesis.
Chapter 3 - Contains a proposal of the solution to the presented problem.
Chapter 4 - Contains the reasoning behind why specific technologies were used in this thesis.
Chapter 5 - Contains an explanation of the implementation of the proposed system.
Chapter 6 - Contains the result of the tests performed on the system.
Chapter 7 - Contains the discussion of the results.
Chapter 8 - Contains the conclusion of the thesis.

Chapter 2
Background Theory

In order to realize a fully functioning access control system several different techniques have been used. In this chapter a brief overview is given of these different techniques.

2.1 Message Authentication Code

A Message Authentication Code (MAC [5]), is used as a way of authenticating a message received digitally. It provides a way of detecting if the message has been changed or tampered with in any way by adding a piece of information to the end of the message. This piece of information has been calculated using the message and a secret key that is shared between the sender and the receiver. The definition of MAC does not specify the means of calculation and a number of schemes has been proposed and implemented, including Hash-based MAC (HMAC [6]) and Universal MAC (UMAC [7]) . A MAC differs from other message authentication schemes such as Digital Signatures, e.g. DSS [8], as it uses symmetric keys instead of asymmetric keys. This leads to a loss of some of the functions of e.g. DSS such as non-repudiation, where a public key of a sender can be used to prove to others that the information was actually signed by him/her. Should a conflict arise in symmetric key cryptography one party can always claim that the information was created by the other as no personal information is put on the message, anyone in possession of the shared secret can create exactly the same message. Using symmetric keys however leads to much less demanding calculations and are therefore suitable for cases where computational power is limited. [5]

2.1.1 HMAC

Hash-based Message Authentication Code (HMAC), is an implementation of MACs that uses cryptographic hash functions to calculate the MAC. Any iterative hash function can be used, e.g SHA-1 [9] or MD5 [10], making it a very flexible implementation with many uses. The message is hashed using the function of choice together with a secret key, a key that has been pre-shared with the recipient of the message. The string of information, the MAC, created by doing this is appended to the message and upon receiving the message the recipient is then able to use the pre-shared key to perform the same hash function on the message to verify

that this gives the same MAC. If it should differ from the one received with the message the recipient knows that the message has changed along the way or that has been generated by an unauthorized party.
The security of HMAC depends both on the choice of hash function and the choice of the secret key. The strongest possible hash function should be chosen in all cases apart from where other criteria dictates otherwise, such as e.g. limited computational power. The key should be chosen at random to prevent key forgery attacks and should be at least as long as the output byte-length of the chosen hash function, e.g. 20 bytes for SHA-1. Using a longer key than the output length will not improve the strength of the function considerably. [6]

2.2 Transport Layer Security

Transport Layer Security (TLS) is a cryptographic network security protocol that consists of two layers, the TLS Record Protocol and the TLS Handshake Protocol, that provides privacy and message integrity over a network. An important property of TLS is that it is application layer independent, meaning that higher level protocols can be layered on top of TLS transparently.[11]

2.2.1 TLS Record Protocol

The TLS Record Protocol is located on top of a reliable transport layer protocol (e.g. TCP [12]) and is what provides the connection with privacy and message integrity.[11]

- Privacy is achieved by encrypting the data to be sent with symmetric keys that are generated, uniquely for each session, from a master secret agreed upon using the TLS Handshake Protocol.
- Message integrity is provided by including a message check in the form of a keyed MAC generated through a secure hash function. This step is optional.

2.2.2 TLS Handshake Protocol

Before a TLS connection is established the communicating parties must perform a handshake with each other. During the handshake the parties authenticate themselves to each other using asymmetric keys[1] that is often part of a certificate (e.g. X.509) and exchange necessary security parameters needed for the TLS Record Protocol. The steps performed during this handshake are as follows [11]:

- The parties exchange hello messages agreeing on algorithms, exchange random values and check if there is an old session waiting to be resumed.
- The parties exchange the cryptographic parameters necessary to agree on a premaster secret.

[1] Mutual authentication is in fact optional, often only server authentication is performed.

- Authentication is performed by the parties exchanging certificates and cryptographic information with each other. This step is optional.
- With the premaster secret and the random values exchanged earlier a master secret is generated.
- Security parameters is provided to the TLS Record Layer.
- The communicating parties verify that they have generated the same security parameters and that no tampering was done to the handshake by an attacker.

2.3 Public Key Certificate

A public key certificate is an electronic document that is used to identify users on a network. The certificate binds a public key to an identity, containing information such as a name and an email address, using a digital signature. Typically this signature will be of a certificate authority (CA), a highly trusted third party, that guarantees that the public key and the identity belong together. [13]

2.3.1 X.509

X.509 is an ITU-T [2] standard in cryptography for Public Key Infrastructure and Privilege Management Infrastructure which specifies, for example, standard formats for public key certificates and certificate revocation lists. The X.509 system makes use of a CA, an entity which issues digital certificates binding a public key to a particular distinguished name. In an X.509 system the validity of the certificate is checked by verifying that it is signed by a CA for that system and also that it has not been revoked. [13]

2.3.2 Certificate Structure

The structure of the X.509 certificate v3 is defined as follows [13]:

- Certificate
  - Version
  - Serial Number
  - Algorithm ID
  - Issuer
  - Validity
    - Not Before
    - Not After
  - Subject
  - Subject Public Key Info
    - Public Key Algorithm

[2] A sector of the International Telecommunication Unit that makes standards for telecommunication

    - Subject Public Key
  - Issuer Unique Identifier (Optional)
  - Subject Unique Identifier (Optional)
  - Extensions (Optional)
- Certificate Signature Algorithm
- Certificate Signature

2.4 Manufacturing Message Specification

The Manufacturing Message Specification (MMS) is a standardized system for exchanging real-time data between networked devices. It defines a set of standard objects such as variables, domains and journals on which operations like read and write can be performed, a set of standard messages to be exchanged and a set of standard encodings for these messages. MMS is designed to function independently, both of the tasks being performed and the developer of the device in question and is therefore the protocol of choice for many makers of industrial networked devices. The task independence means that e.g. production data is accessed in exactly the same way as energy management data, which is something that greatly streamlines the process of adding additional devices such as sensors to an existing network. [14].

Chapter 3
Proposed Framework

As proposed in [2] the access control system will be implemented according to the following description. For the purpose of this report and for the simplification of comparisons with other similar systems the system implemented and tested in this project will be referred to as **Trinity**.

3.1 Proposed Solution

A client wishes to gain access to a service provider (SP). This can be any machine capable of delivering a service to a user but in this project the SP is considered as having constrained resources in the sense that it should focus all its memory and processing power on its designated tasks. As a result of the lack of available resources the SP is assumed to be unable to run any kind of additional heavy processes such as the parsing of user databases. This is however necessary in order to perform access control evaluation on a user, which is why in this system there exists a third party known as an Access Control Server (ACS). This ACS is trusted by the SP hence it is possible to let the ACS perform the access control on behalf of the SP. A graphical overview of a typical scenario is given in figure 3.1.

0. Secret keys are distributed to the ACS and SP creating a trusted relationship between the two. The ACS is also configured with the access control policies of the SPs it services.
1. A client sends a request to the ACS detailing which SP and which service on this SP it wishes to gain access to.
2. The ACS checks the client request and if access is granted responds by sending a ”token” to the client. This token contains a representation of the ACS’s access control decision. The tokens integrity is protected by a Message Authentication Code (MAC) created by the ACS using a secret known only by the ACS and the SP.
3. The client sends a request and the token to the SP.
4. The SP verifies that the token has come from the ACS using the aforementioned shared secret and checks that the token was indeed issued to the requesting client and for the requested resource. Finally the SP responds by giving the client access to the requested service.

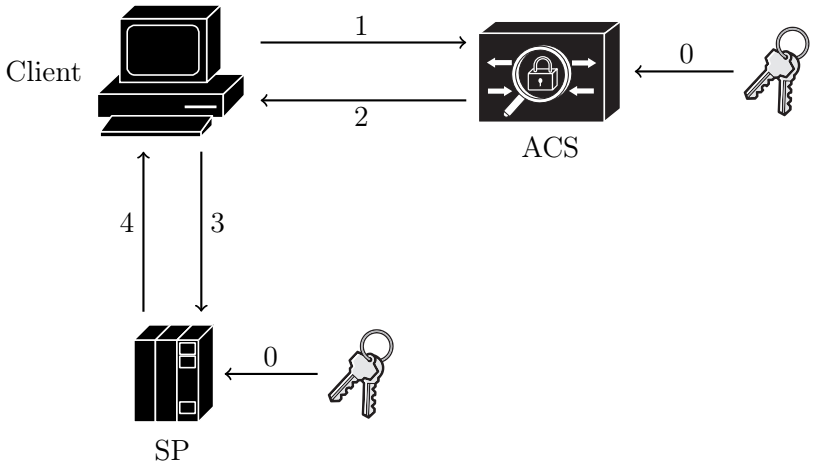

Figure 3.1: A graphical overview of Trinity

3.2 Design Goals

When designing Trinity the following aspects will be considered.

- The size of the token should be kept to a minimum, both during transmission and in storage on the SP.
- The user experience should remain unchanged, an end user should not have to learn how to use Trinity.
- Token processing on the SP should be executed as efficiently as possible with little to no impact on the normal operating procedures of the system.
- The impact on the size of the firmware of the SP should be minimal.
- Token transmission from the Client to the SP should be done with as little impact on the SP as possible.

3.2.1 Security Considerations

The security considerations of Trinity match those presented in [2] and can be summarized by the following points.

- Trinity aims to protect both the services given by the SP and the SP itself.
- There must exist a trusted relationship between the SP and the ACS.
- The shared secret between the SP and the ACS must be protected.
- The ACS is considered a single point of failure of the system, should this be compromised Trinity can no longer be trusted. The ACS must therefore be very well protected against all sorts of attacks.

- In order to properly authenticate clients the ACS and the SP must be able to identify them.
- To protect against eavesdropping and traffic manipulation the communication in Trinity should be encrypted and integrity protected.
- The token must be integrity protected in order to guarantee correct authorization.
- The tokens being sent to the SP should be stored securely on the machine.

Chapter 4
Choosing Technologies

4.1 Hardware

The hardware used in this project is an 800xA controller provided by ABB. The hardware specifications of this product are a trade secret but for all intents and purposes it can be considered to have considerable resource constraints. Not in the traditional sense, since it does have adequate processing power, but more in the sense that all its resources should go to running the applications on it which leaves a very constrained share for security measures. The controller has built in support for communication encryption and client authentication using SSL/TLS by using the OpenSSL library since this has been implemented by ABB in a previous research project. This controller will serve as the resource limited service provider (SP).

4.2 Software

4.2.1 Control Builder

This is a Windows application written by ABB that is used to program and perform general operations on their controllers. Originally the communication between the controller and the Control Builder offered no authentication or authorization in any form. However for this thesis a Control Builder version with TLS-support will serve as the client.

4.2.2 Access Control Server

There exist many different implementations of an Access Control Server[1]. Not only would implementing one of these systems add no scientific value but it would also be infeasible to do so due to the limited scope of this thesis. Instead a simplified version will be written that includes only the options needed for this thesis. The interface between the client and the ACS will however have full functionality so that the simplified ACS can be easily replaced by a real ACS.

[1] DIAMETER AAA[15], RACF[16]

4.2.3 OpenSSL

OpenSSL is an open source cryptographic library that implements full support for the SSL (v2/v3) and TLS (v1) protocols. The project is volunteer-driven and can be used freely for both commercial and non-commercial purposes[17]. It is the most widely used cryptographic library on the Internet and offers both full support of TLS and a large library of cryptographic functions, including HMAC. This added to the fact that the hardware provided by ABB already includes support for this library makes it ideal for this project.

4.3 Security Considerations

4.3.1 Encrypted Communication

An important feature for Trinity to provide proper security is that the communication between the different parties can be done securely, i.e. that the communication is encrypted in some way. There are several reasons as to why this is important. An attacker is able to listen to traffic that is not encrypted, known as eavesdropping, thereby possibly finding high-value information such as the type of services a SP is able to perform (useful when doing searches for possible high-value targets where e.g. the service ”SetReactorTemp” would make the attacker able to guess that the SP is used in a nuclear power plant). This also means that an attacker can get his hands on a valid token. These tokens might later be used to impersonate an authorized client, which would be very bad as this enables an attacker to actually interact with the system thereby breaking the actual access control mechanism.

There exist many technologies for encrypting communication over networks where one of the most widely used is TLS. TLS is as mentioned above supported in the 800xA controller through OpenSSL and will therefore be used in this thesis.

4.3.2 Trusted Relationship SP-ACS

As mentioned in the proposed solution there needs to exist a trusted relationship between the Service Provider and the Access Control Server. This is achieved by letting them share a secret, which here consists of a pre-shared key. As this key is such an important requirement for the system it should be protected by various layers of security in order to prevent it from being stolen. This is however considered as not part of this thesis and is left for future implementations.

The trusted relationship exists partly to ensure that the token used by the client to access the SP is in fact issued by the ACS. This is done by adding a MAC to the end of the token. This MAC also ensures that the token is integrity protected. The reason for using a MAC over e.g. Digital Signatures is explained further in section 2.1 but involves computational power. There exist many different types of MACs and the one used in this solution is a SHA-1 based HMAC, mainly because

of the built in support for this scheme in OpenSSL[2]. The choice of the SHA-1 hash algorithm sets the length of the pre-shared key to 160 bits, the output hash length of SHA-1.

4.3.3 Securing Identities

In an access control scheme the identities of the users needs to be well defined and impossible to fake. If the only criteria had been that each user should have a unique identity any naming or numbering scheme could have been used in this solution. However, since each by the ACS issued token has to be linked to a user identity, each user identity has to be linked to the actual user and consist of something that is not possible for an attacker to modify in order to use a token that was not intended for him/her. One might argue that since the communication is encrypted there is no way for an attacker to intercept a token anyway, so why bother with a complex naming scheme? This is of course true, but on the Internet today this extra layer of security can still be considered necessary as an encryption can be circumvented.

A way of ensuring that the identities can still be used even if the encryption is broken is to use the X.509-certificates from the TLS-connection between the user and the ACS as user identities. As these are issued by a trusted Certificate Authority they are very hard to fake and thus fulfill both of the presented requirements. Should the encryption of the communication fail the system is susceptible to eavesdropping, but the fact that the attacker cannot fake his/hers own identity ensures that no actual modification of the traffic is possible, keeping the core function of the system intact. The X.509-certificates contain a parameter called Distinguished Name, which can be extracted using OpenSSL functions. These are what will serve as client identities in this thesis.

4.3.4 Protection of Involved Nodes

The nodes that are most prone to become targets of a coordinated attack are the Service Provider and the Access Control Server. These must therefore be well protected by different levels of security. Possible attacks on these nodes include the use of a Denial-of-Service attack [18] in order to overload them, the extraction of secret information stored on them and, in the case of the ACS, physically disconnecting the node in order to shut down the entire system.

In order to initiate communication with any of these nodes a client must first set up a valid TLS-connection with it, which gives an inherent protection against DoS-attacks that center on sending multiple requests or fake tokens. If a client is not able to set up the TLS-connection, which implies that the client has not been authorized by a trusted Certificate Authority, it will not be able to send neither requests nor tokens to any node. Apart from this no other protection has been implemented against DoS-attacks.

[2]Note that SHA-1 is no longer considered secure, other possibilities are examined in chapter 8.

Extraction of secret information from the nodes such as pre-shared keys and user access data would lead to the whole system being compromised, as an attacker would then be able to generate valid tokens and gain access to any service present on the SP. However, protection of this information on these nodes is not considered a part of this thesis and is left for future implementations.

Being a single point-of-failure the ACS would be considered a high-value target. The implementation of a secure ACS is however not considered a part of this thesis and it is henceforth assumed when discussing the ACS that, since it is not resource constrained, it is possible to implement sufficient security measures on this machine.

4.4 Summary

- An ABB 800xA controller will serve as the Service Provider.
- The Control Builder application from ABB will serve as the Client.
- An Access Control Server will be written solely for this thesis.

Based on the security considerations mentioned above the following techniques will be used:

- TLS to encrypt communication between nodes. OpenSSL to implement TLS.
- The Distinguished Name from the X.509-certificates as identification of clients.
- HMAC to ensure message integrity of token.

Chapter 5
Implementation

The system consists of 3 parties: the Service Provider (Controller), the Client (Control Builder) and the Access Control Server. An overview of the implementation is given below.

5.1 Delimitations

The controller and the CB communicate through an application level protocol called MMS that is layered on top of TCP. Due to time limitations and the fact that the system is only a prototype it was decided that only two different MMS resources were to be protected by the authorization system, namely the initiate Upload and Download requests.

For the purpose of this project there is no need to implement the ACS as a realistic access control engine, since this adds no scientific value and such products are well known. Instead the ACS simply parses a text file containing a few identities with their respective authorization as a simulation of an actual access control engine. The interface between the client and the ACS is however built with full functionality so that the simplified ACS can easily be replaced with a real access control engine.

5.2 Communication between Involved Parties

Many choices made in the implementation of Trinity are based on the structure of the existing communication between the Control Builder and the controller. The following section describes the communication between a controller and the Control Builder where TLS has been implemented.

All communication between the two parties is done using the MMS protocol. For each MMS request made by the the Control Builder to a controller a full TLS handshake is performed, the MMS request is sent and the TLS connection is torn

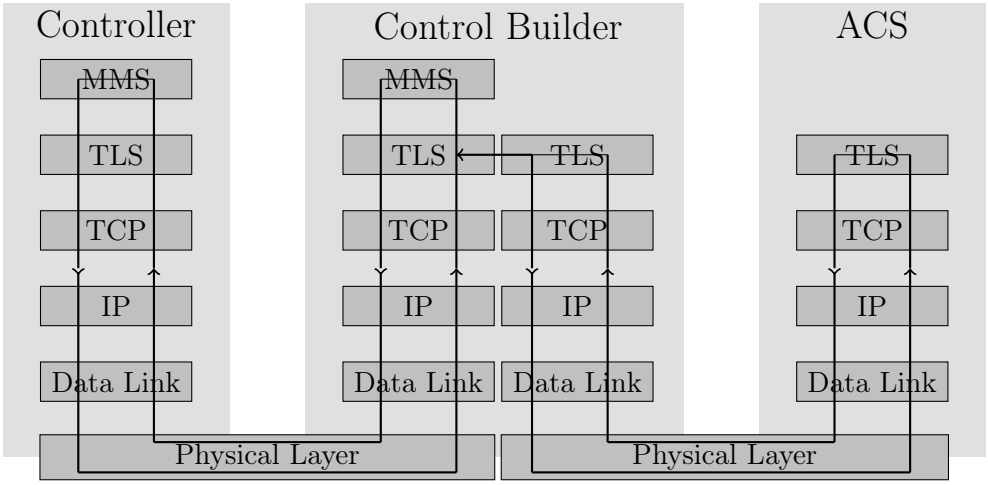

Figure 5.1: The communication stacks of the involved parties

down[1]. Should the request consist of multiple MMS messages a single TLS connection is still made. Wishing to preserve as much of the original design as possible Trinity will have to work using this communication structure.

The Control Builder, which serves as the client, will apart from being able to communicate with the controller also have to be able to communicate with the Access Control Server. Only being used to request and send tokens, this communication requires no additional upper layer protocols above TLS and will therefore be implemented separately from the rest of the communication done by the Control Builder.

Figure 5.1 depicts how the different communication stacks interact in all involved parties.

5.3 Token

The structure of the token implemented in Trinity is based on the one presented in [2] and consists of a string with the following characteristics, further described in table 5.1:

```
{SN:xx;IS:xx;SI:xx;TA:xx;VF:time_int;VU:time_int;AC:11111111}HMAC:xx
```

[1]This actually depends on the operation mode of the Control Builder. Should it be ”online” with a controller the TLS connection is not always torn down when a request has been processed, this varies between different requests. Being ”online” is however not considered as the normal working scenario of Trinity as it is a time consuming process used mainly when doing more substantial controller configuration.

Table 5.1: Token description

Encoding	Description	Size (Bytes)
SN	Sequence Number	8
IS	ISsuer	10
SI	Subject Identifier	200
TA	TArget	15
VF	Valid From	10
VU	Valid Until	10
AC	ACcess	1
HMAC	HMAC value	20

Each new value begins with the description of the value followed by a colon and ends with a semicolon. The entire token is surrounded by braces in order to easily be able to extract the parts of the token upon which the HMAC should be calculated. The decision to have the token be a string value is based partly on the fact that the 800xA controller implements a string library in its original configuration, but also that it increases readability and thereby eases debugging. Other possible structures are discussed in chapter 8. The Sequence Number of the token is used for token revocation, although no such functionality is implemented in Trinity. The Issuer is the identity of the ACS and can be used to quickly verify if the token is issued by a valid ACS, something that will otherwise reveal itself upon calculation of the HMAC. The issuer can also be used if there exist multiple issuers in order to know which pre-shared key to use when calculating the HMAC. The Subject Identifier is the identity of the client to which the token is intended. In Trinity the Subject Identifier is the distinguished name of the X.509 certificate used by the client to set up the TLS connection with the ACS and later with the controller. As the length of the distinguished name can vary between certificates a redundancy has been built in that allows for names to be as long as 200 bytes. The Target is the IP address of the controller on which the token is intended to be used. The values Valid From and Valid Until present the validity time of the token in UNIX time. The Valid From value presents the possibility to give out tokens ahead of time. Possible use cases for this might include a system engineer getting access to perform system updates only after a certain time has passed to allow users to complete their current work.

The design of the Access part of the token has gone through a number of different iterations. It bases itself on how token exchange is performed between the involved parties. If a unique token is requested and sent to the controller for every request sent by the Control Builder the token has to grant access to a single resource. This was the initial plan, however because of the structure of the existing MMS communication this proved to be an inefficient solution. It was instead decided that a token should be sent once and then stored on the controller to be retrieved and checked every time the Control Builder requests access to a protected

resource. The Control Builder will then in turn remember if a token has been sent in order to avoid sending unnecessary tokens. To handle this a single token can grant or deny access to a number of resources. In Trinity the total number of resources is eight as they are represented as particular bits in one byte. This number is however very easily increased should it be needed for future implementations. If access is granted to a resource the bit in question is 1, otherwise it is 0. The HMAC is calculated on the part of the token surrounded by the braces, including the braces themselves, and appended to the end of the token string.

5.4 Access Control Server

The ACS is a written entirely for the purpose of this project. It has the properties of a basic TLS server that waits for an incoming TLS connection from a client. During the TLS handshake the client and server authenticate each other using X.509 certificates that were generated by the OpenSSL library and signed by a certificate authority. After the connection is established the server waits for a token request from the client. When the request is received the ACS parses through a text file containing all authorization information looking for the access rights for the identity given by the certificate. If the CB has any access rights on the controller a token is created and integrity protected with a HMAC, created using a shared secret between the ACS and the controller, and sent as a response to the client. If no access rights can be found the connection is terminated and the client receives no access token.

5.5 Client - Control Builder

For this implementation of Trinity the Control Builder will serve as the Client that is trying to get access to a protected resource in a Service Provider (Controller). At the start of this project there was no support for access control implemented in the CB that could be built upon. Thus the access control system needed to be implemented from scratch and integrated into the existing communication protocols of the CB and the controller. When implementing the access control system it was important to remember that for the purpose of this thesis the CB, unlike the controller, is considered to have unlimited resources. This means that as much of the workload as possible should be distributed to the CB and not the controller, and that this work should be done before a connection is established with the controller, affecting the controller as little as possible. One important aspect of limiting the workload on the controller is to avoid unnecessary transmissions of tokens. This is accomplished by letting the CB perform simple checks on the plain text part of the token before it is sent and also having it keep track of what tokens the controller already has that belongs to the CB in question. With respect to all this, the functions that needs to be implemented on the CB are:

- Retrieval of a token from the ACS.
- Storage of that token on the CB.

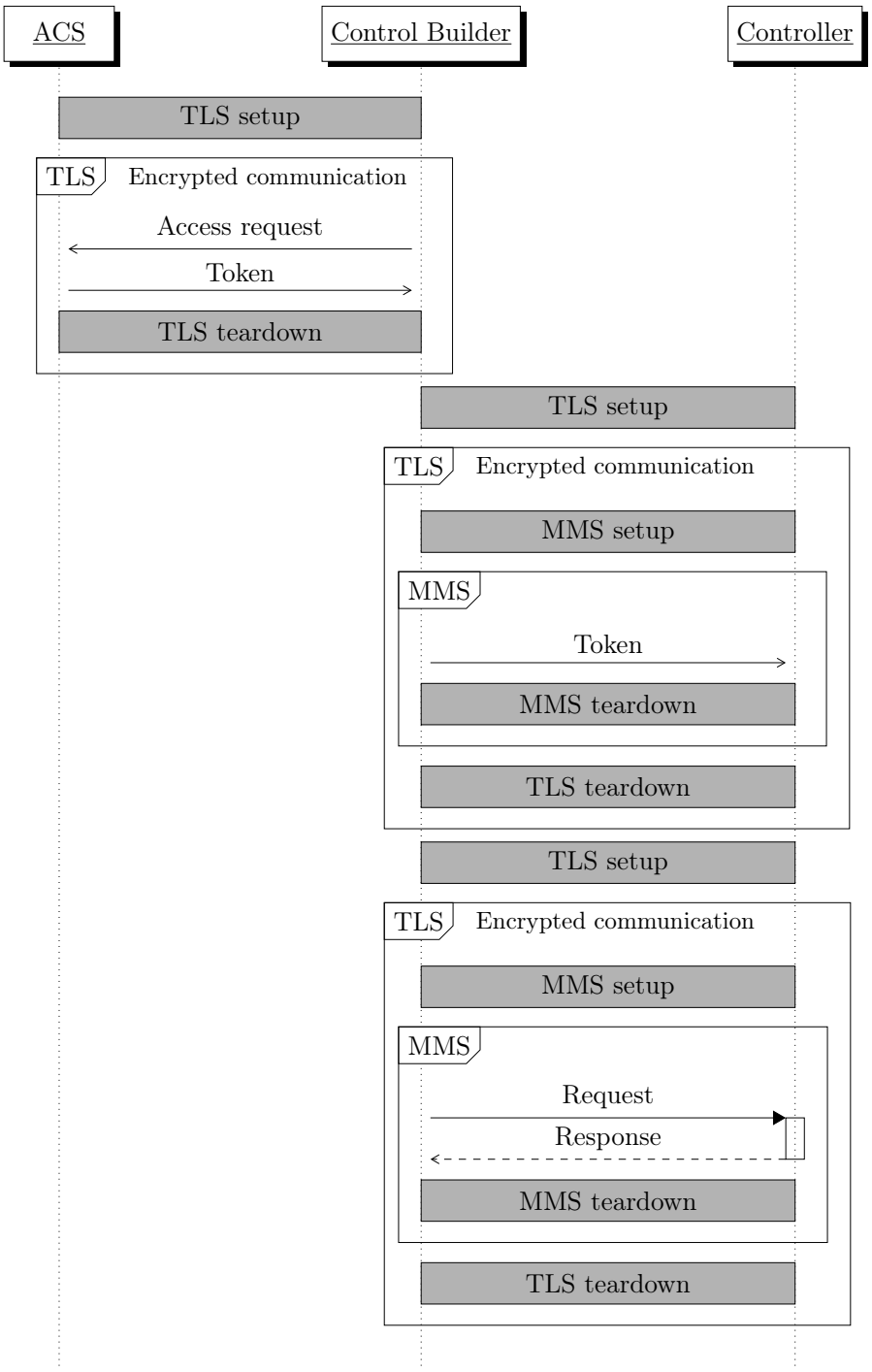

Figure 5.2: Sequence diagram of communication in Trinity

- Check if the token should be sent to the controller.
- Sending the token to the controller.

5.5.1 Retrieving Token from the Access Control Server

To be able to connect to the ACS from the CB the original protocol layers on the CB were circumvented as they rely on the communication being done using the MMS protocol, whereas the top layer protocol of the ACS is TLS. Therefore a separate communication stack was implemented on the CB that has TLS as the top layer protocol. When the CB needs to retrieve a token it generates a token request that consists of a string that contains the IP address of the controller that the CB want to gain access to. The characteristics of this request can be seen below.

```
{TA:xxx.xxx.xxx.xxx}
```

After the request is generated a TLS connection is established between the CB and the ACS and the request is sent. If any kind of access is granted a token is generated in the ACS containing all the authorization information that the CB in question has on the controller that it wanted access to. If no access is granted the ACS terminates the connection and an error message is displayed for the user on the CB.

5.5.2 Token Validation

When a token is retrieved from the ACS it is reasonable, although not mandatory, that the CB performs some checks to see if the token received should be sent to the controller. The check implemented in this project is that the CB verifies that the token has become valid and that is it not yet expired. If the token should have expired it is to be discarded but if the token is not valid yet it is stored in the token storage to avoid additional retrievals from the ACS, since a new retrieval will only result in the same token. The not yet valid token is not sent to the controller until it is valid, instead when the CB tries to access a protected resource with a token that is not yet valid, a pop-up will appear on the CB telling the user when access will be granted. This means that no token will be requested from the ACS or be sent to the controller until the token has become valid. It is important to note that there is no need to check if the token grants access to the controller in the CB since the ACS only sends a token if any kind of access is actually given. There is also no to need check which access has been given. This is due to the fact that the properties of the resource request is unknown to the operating layer of Trinity at the time when the MMS connection is established. Hence it is not possible for the CB to check if it has access to the resource it is going to request before the MMS connection is set up. The request call could still be blocked from the CB but since the MMS connection is set up anyway and the connection is by far the most resource consuming operation it was deemed unnecessary to implement such a check on the CB.

5.5.3 Token Storage

Storing tokens not only on the controller but also on the CB allows the CB to keep track of the tokens already stored on the controller concerning the CB in question, thus avoiding unnecessary retransmissions of tokens. The tokens are saved in a hash table with the token request as the key and a struct, containing the entirety of the token and also a flag if the token has been sent or not, as the value. The reason for saving the tokens in a hash table is that this structure provides the needed functionality, as one value can be mapped to another. An implementation of a hash table already exists in the original firmware. Saving the entire token drastically reduces the amount of retransmissions needed between the ACS and the CB. Now if the token lists on the CB and the controller should become out of sync and the CB thinks that the controller has a token that has in fact been lost, the CB can simply retrieve the token from its own list and retransmit it instead of requesting a new one from the ACS.

5.5.4 Token Transmission

An intricate part of Trinity is the token transmission between the involved parties. The first thought was to send it using only TCP messages, not involving the higher level protocol MMS at all, as it was considered effective to be able to deny access as early as possible in the connection. This proved a difficult task since the TCP layer is heavily integrated with the MMS protocol and the TCP buffer is not really analyzed before it reaches the application layer. Therefore it was deemed infeasible to make any sort of changes to the TCP buffer without affecting the upper protocol layers. The second and more successful idea was to use an existing functionality called MMS variables in the MMS protocol to send the tokens. These variables can, after being defined in both the CB and the controller, be easily transmitted between the two devices which is exactly the functionality sought after. The transmission is done by using an existing functionality in the CB that sets up an MMS connection with the controller and sends the variable as a string. This way of transmitting tokens was deemed to be the most effective since it required very little changes to the existing code. A sequence diagram showing a typical connection scenario is shown in figure 5.2. Also shown here is how the CB before connecting to the controller requests an access token from the ACS.

5.6 Service Provider - Controller

As mentioned before a 800xA controller will serve as the resource constrained service provider in this thesis. Originally not having any support for access control over a network [2] the difficulties lie in finding a way to integrate Trinity into the existing system with as little impact as possible on the way it normally operates.

[2] Access control does exists in the form of a physical key that can be used to ”lock” certain functions on the controller. This access control is however not user specific in any way.

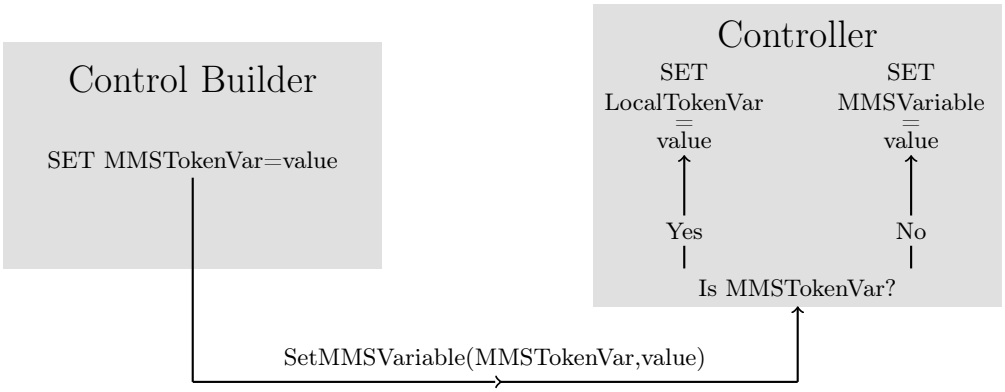

Figure 5.3: Token reception

The functions that need to be implemented are:

- Receiving a token from a client and storing this token on internal memory.
- Validating tokens.
- Parsing requests from a client in order to grant or deny access.
- In case of denied access: Replying the client with reason for denial.

5.6.1 Receive Token

As mentioned before MMS variables will be used to transfer the token from the CB to the controller. An MMS string variable was therefore created on the controller that is accessible from the CB and when sending a token the CB connects to the controller and sends an MMS request to change the value of this variable to that of the token. Note here that access to this token variable must not be and is not protected by Trinity. The controller monitors the variable and when receiving the command to set it to a certain value, the value is instead stored in a different, non-MMS variable. This way the token is never actually stored in the non-protected MMS variable. A graphical overview of this procedure can be seen in figure 5.3.

5.6.2 Token Validation and Storage

An important consideration when implementing the token validation is that should this turn out to be a demanding task, it must not hinder the controller in any way by preventing it from performing other more important tasks. With this in mind it was quickly decided that the token validation should not occur directly after receiving the token, as a client has no way of knowing which tasks the controller is currently executing and therefore does not know when it can send a token without disturbing the system. The controller performs tasks by periodically polling a list of subsystems in the order that matches their priority. By adding a subsystem to the end of this list, thereby giving it the lowest priority, it is possible to have the controller perform the token validation only when tasks with higher priority

have been completed. This subsystem then has to check if a token has been stored internally and if so, perform the necessary checks in order to validate it. The validation steps performed in Trinity are:

1. Check if the token is valid in time by using the internal clock.
2. Check if the token is issued by a trusted ACS and which pre-shared key to use[3].
3. Check if the target of the token matches the intended target, i.e. itself.
4. Check if the identity of the connected client matches that in the token.
5. Calculate the HMAC of the token and verify with the value present at the end of the token.

The order in which these checks are performed is of importance. The most likely error scenario is considered to be that a client sends a token that has expired [4]. Therefore the first check should be if the token has expired or not. The remaining checks are performed in an order of decreasing priority up until the calculation of the HMAC. This is a very important check, however it is also theoretically the one that uses the most computing power. Should a token be invalid in any of the other checks it is good to be able to deny access before having to calculate a then unnecessary HMAC. If a token passes all checks it is to be stored on the heap of the controller in a hash table. Storing the entire token would however be unnecessary; only the most essential parts need to be stored in order to save memory space. The Subject Identifier (SI) from the token is used in the hash table as the key that links to the expiration date and the access parameters of the token. When later retrieving a token from the table in order to check the access of a client, the controller can check if the token has expired and if the client in question has the requested access. If the token is expired it is removed from the hash table.

5.6.3 Authorization

The controller monitors all MMS requests and when a CB requests access to a protected resource the controller uses the distinguished name from the TLS-connection set up with the CB to retrieve a stored token from the hash table. Should a token not be found or should the retrieved token not grant access to the requested resource the MMS request is aborted and an error message it sent to the CB using existing MMS functions with the addition of the messages *TokenRequired* and *TokenDenied*. If access is granted the request is handled as it normally would with no further changes to the framework. This way the impact on the ordinary operations of the controller is kept to a minimum.

[3]This functionality is not used in this implementation as there only exists one ACS.

[4]This considers only benign access attempts, attacks using e.g. fake tokens is also an issue but is not the primary concern of the framework.

Chapter 6
Evaluation of the Implementation

To measure the success and usefulness of Trinity it was important that measurements were taken from all areas where the 800xA controller is considered to be resource limited. The controller is resource limited in the sense that is has limited memory and most of all that it is very sensitive to time delays. The reason for the controller being time delay sensitive is that it is very important for it to be able to run its designated tasks, that are very often in charge of running expensive and important machinery, without any interruptions. Hence it was of utmost importance that the performance tests of this implementations were performed on a system running a real life application, limiting the resources available for additional computing, thus giving a better understanding of how Trinity works in a resource constrained environment.

No measurements were taken from the Control Builder and the ACS as their performance is of no interest to this project. Both of these parties are considered to have unlimited resources and therefore it is not important how long it takes for them to perform an operation or how much memory this access control system uses on them. Both the ACS and the CB can devote all of their available resources to the access control system since neither of them run any high priority applications like the controller does. Therefore the controller is the only part of this system where it was measured how resource consuming this access control system is.

6.1 Method

6.1.1 Test Setup

There were two different test setups used during these tests. One that consists of an empty controller running no application and one that consists of 3 interconnected 800xA controllers running an emergency shutdown application from a real life oil rig. When running the aforementioned application the controllers perform several different operations as well as communicating amongst each other, mimicking a real life situation for these controllers and thereby making it a suitable test environment for this project. The tests were done with three different firmwares for comparison:

- The *Original* firmware, performing no authentication, encryption or authorization.
- The *TLS* firmware, performing authentication and encryption but no authorization.
- The *Trinity* firmware, performing authentication, encryption and authorization.

6.1.2 Memory Usage

There are two types of memory on the controller that are of interest for this project, the RAM and the ROM. The ROM is where the firmware is stored and what affects the memory usage in it is how much bigger the firmware has gotten from the addition of the access control system. The RAM is where the heap and stack is allocated to, meaning that the program will dynamically allocate memory there during run time.

The effect on the memory usage in the ROM by the addition of Trinity was measured by simply looking at the size of the compiled code with and without it. This test was performed because it is important to know how much size the access control system adds to the original source code. In the controller the ROM is a constrained resource and therefore it is important that the implementation of the access control system on the controller is kept short and concise.

During the firmwares run time, memory is dynamically allocated and de-allocated to the RAM of the controller. Since RAM also is a limited resource on the controller it is important to measure how the RAM usage differs when the access control system is turned on and off. The heap and thus the RAM usage was checked using the "Get Heap Statistics" option in the CB which retrieves a list of what elements are allocated on the heap and how big they are. The values that are of interest from this list are how much space the token takes on the heap. This test is performed on a single empty controller since the heap space the token uses does not vary depending on the application the controller is running.

6.1.3 Token Processing

To find out the impact the different token operations performed in Trinity has on the system the time it takes for the controller to perform these different operations is measured on an empty system running the Trinity firmware. An empty system means that it will not abort the operations performed in Trinity to execute some other operation with higher priority, meaning that the time measured will be solely caused by Trinity itself. Only the execution time of the operations will be measured, meaning that no token transmission time is included in the measurements. The operations in question are *validating a token*, *storing a token on the heap* and *retrieving a token from the heap*. The validation will also be measured in cases where the token is not valid and therefore rejected. The invalid cases are where tokens contain an invalid subject identifier or are expired. The reason for

Table 6.1: Size of firmware in different configurations

Firmware conf.	Size [KB]
Trinity	2181
TLS	2177
Original	1588

choosing these particular cases is that the expiration check is the first one done in the validation process and the check of the subject identifier is the one done before the calculation of the HMAC, which is the final step. A mean value of 100 measurements will be presented.

6.1.4 Accessing Protected Resources

To see how different firmwares affect the workload of a real system the aforementioned emergency shutdown setup was used for this test. Measurements were taken of the total TCP connection time when accessing different protected resources on the controllers from a CB. This value gives a good insight of the workload, as a connection is set up upon the first request from the CB and not torn down until all operations have been performed. Four different resources were accessed ten times each on all three controllers for every firmware. In the case of the Trinity firmware a new token was sent for every request to also measure how much the token processing affects the system. The resources in question all use operations that have been restricted by Trinity and are called *Show downloaded items*, *Show firmware information*, *Show MMS connections* and *Show MMS variables*.

6.2 Results

6.2.1 ROM

How the file size differs between the original firmware, the firmware with TLS and the firmware with TLS and Trinity can be seen in 6.1.

6.2.2 RAM

It was found that one token takes up 0.00398 % of the available heap space on a controller running the Trinity firmware.

6.2.3 Token Processing

The result of the token operation measurements can be seen in 6.2 and 6.3.

Table 6.2: Token processing, mean of 100 measurements

Task	Time (ms)
Validating token	1.673
Storing token on heap	0.344
Retrieving token from heap	0.085

Table 6.3: Token validation, mean of 100 measurements

Token status	Time (ms)
Valid token	1.673
Invalid subject	0.122
Expired token	0.100

6.2.4 Accessing Protected Resources

The results when accessing the protected resources on all three controllers with different firmwares can be seen in figures 6.1, 6.2 and 6.3.

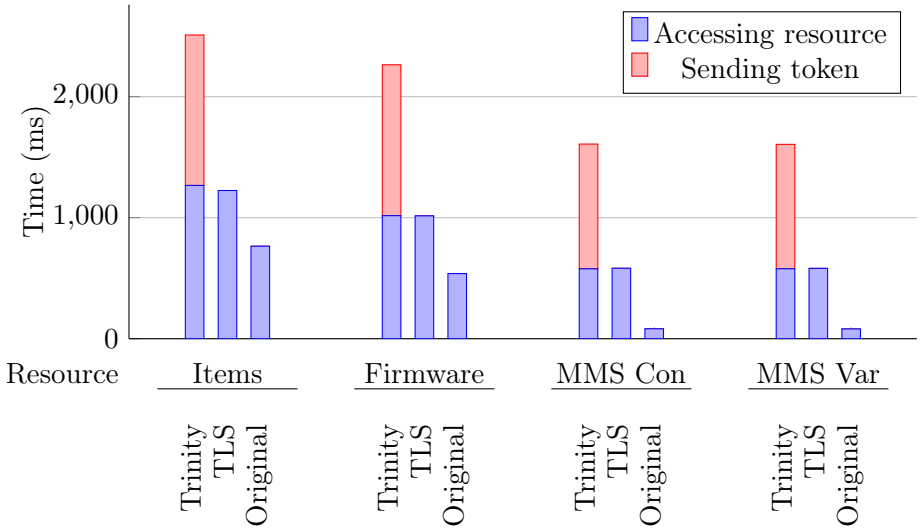

Figure 6.1: Controller 1 - Connection time

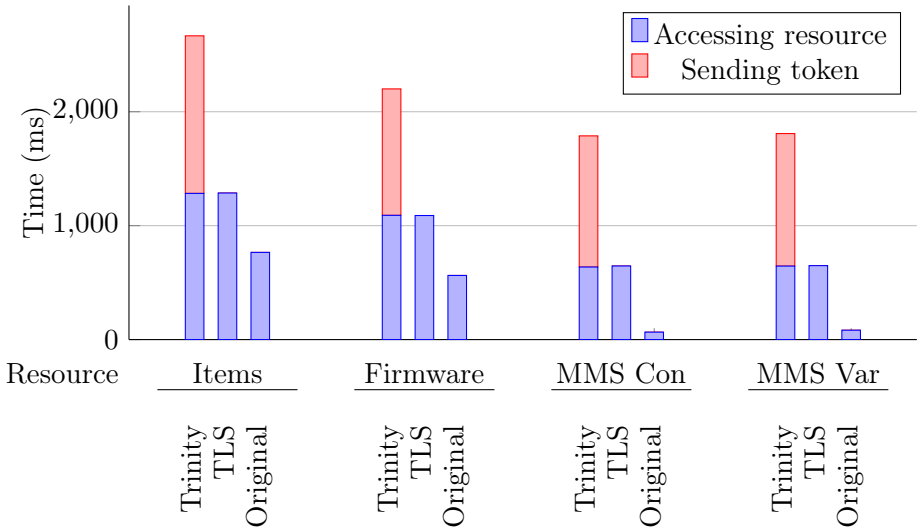

Figure 6.2: Controller 2 - Connection time

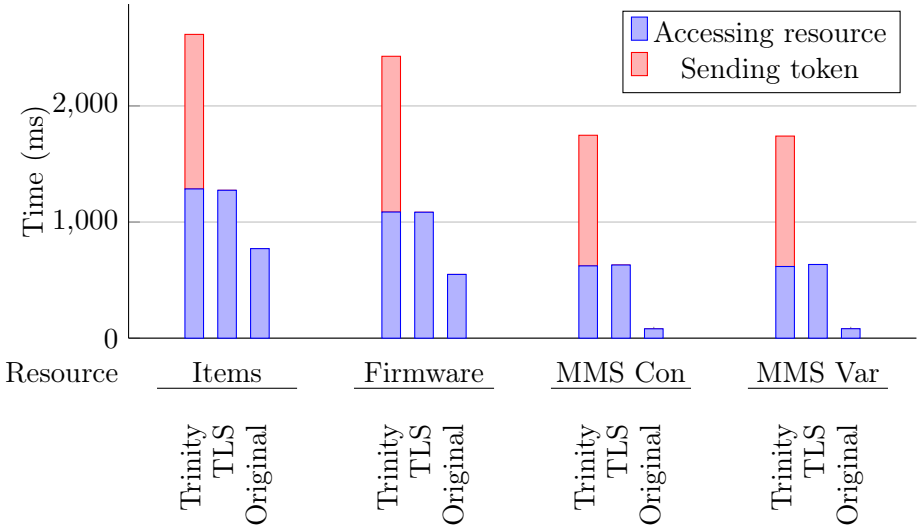

Figure 6.3: Controller 3 - Connection time

Chapter 7
Discussion

7.1 Memory Usage

7.1.1 ROM

What can be seen in table 6.1 is that the smallest firmware is the original, which is an expected result. The rather big increase in size from the original to the TLS firmware, 37%, is mostly due to the inclusion of the OpenSSL library in the TLS firmware. The addition of Trinity to the TLS firmware does however not cause that big of a change in size, only an increase of 0.2%. This is also an expected result as the amount of code added in order to implement the functions needed for Trinity is rather small.

It was always assumed that adding authentication in the form of TLS would affect the system in a big way. This is seen when looking at the 37% increase in firmware size with the addition of OpenSSL. However, since the actual size added by Trinity is very small, the design goal of having the firmware be as small as possible is considered met.

7.1.2 RAM

As can be seen in section 6.2.2 a single token uses 0.00398% of the available heap space. This means that a total of 25125 tokens can be stored on the heap with the assumption that nothing else affects the heap during run time. We would like to suggest that this is more than enough. A single token contains all the access information of an authorized client, which means that the number of tokens in a running system equals the number of clients that has been authorized on that particular system at any given time. In the case of a control system working in a real life environment it is in our opinion likely that the number of authorized clients is no more than 100, not even close to the theoretical limit of over 25000. Note also that Trinity makes use of certificates on the client machines as opposed to personal user login which means that one authorized client does not equal one authorized user; there can be many users working from a single client. Had the token instead been designed to give access to a single resource, the number of tokens stored on the system would have been much higher as every client would need as many tokens as the number of resources it has access to. Although the tokens

designed in that way could have been smaller, we believe that what is gained in size reduction is no way near as significant as what is gained in the current design by decreasing the number of tokens.

Based on this reasoning, the design goal stating that the total size of the stored tokens should be kept to a minimum is therefore considered met.

7.2 Token Processing

When disregarding the token transmission and the connection that is set up for this purpose, table 6.2 shows the total time needed by the controller to perform the operations processing the token. We can clearly see that the most demanding operation is validating the token, which takes 1.673 ms, compared to storing the token on the heap, 0.344 ms, and retrieving the token from the heap, 0.085 ms. This is an expected result, as the validation contains calculation of a HMAC, the operation that was initially assumed to be the most resource consuming when it comes to token processing. Storing a token takes longer than retrieving a token from the heap. One might assume that these values should be roughly the same, since a hash table is used to store tokens and an inherent property of a hash table is that inserting and searching the table both have a complexity of O(1). The reason for this difference is that when storing the token it is not only inserted into the hash table but also formatted into a ”Hash entry” containing only the most vital values.

Table 6.3 shows the duration of a token validation depending on the state of the token. When a token is denied due to the reasons shown in the table, it takes a considerably shorter amount of time to perform the operation. To deny access to a token that has expired takes the shortest amount of time, 0.100 ms. As this is the very first validation step it should also be able to reject tokens the fastest, which we can see is the case. A number of other steps are performed ending with checking if the subject matches that of the connected client. However, it can clearly be seen that these steps are done very rapidly, since it only takes 0.022 ms longer to deny access should the subject be invalid. The validation steps up until now are checks of the authorization information of the token. The next and final step is to verify the message integrity of the token, which is done by calculating a HMAC. This was initially assumed to be the most resource consuming step of the validation and that is the reason it was placed last in the validation process. The results from the table verify these assumptions since it takes 1.551 ms, 92% of the total validation process, to calculate the HMAC and thereby completing the validation. We feel that this proves the statement that the order in which the different validation steps are performed is important.

7.3 Accessing Protected Resources

What we see in figures 6.1, 6.2 and 6.3 is the total duration of the TCP connection on three different controllers when accessing different protected resources on the controller from the client. These tables give a good insight into how the workload differs depending on the firmware of the controller. As can be seen in all figures the original firmware is the least time consuming, i.e. has the smallest workload. This is an expected result as both other firmwares add functionality to the system and removes nothing. The interesting results appear when looking at the TLS firmware. In all cases the increase in time compared to the original firmware is quite significant, around 500 ms. This increase must be the result of the addition of the TLS protocol to the TCP connection since this is the only thing added to the original firmware. This tells us that the TLS protocol is very costly to implement, something that we assumed would be the case from the beginning.

When looking at the Trinity firmware we see two different columns, the lower of which shows the connection time when accessing a protected resource and the upper of which shows the connection time when sending and verifying a token. The sum of these represent the scenario where a client connects to a controller for the first time and has to send a token to be given access. This total connection duration is significantly larger than that of the TLS counterpart, often doubling that value. However, this scenario only occurs when a client connects to a controller for the first time or when a token needs to be re-sent[1], a very rare occurrence. All other resource requests after this initial connection have a duration of that seen in the lower column. Comparing this one to that of the TLS firmware shows a different result. In all of the cases the increase in connection duration is negligible, giving the result that the impact Trinity has on the controller in normal use cases is in fact negligible. In table 6.2 we can see that retrieving a token from the heap and verifying the access of a connected client takes 0.085 ms in an empty controller. This token retrieval is the only operation performed by Trinity in cases where no token is received from the client and even though this value might be a bit higher in a controller simultaneously performing other tasks, it still shows that it is not a demanding operation for the controller to perform, further verifying this result.

The results seen when sending and verifying a token is based on the fact that a separate connection has to be made in order to send the token. This is due to the design of the existing communication framework and for this thesis it was not feasible to look at ways of circumventing this. However, it might be possible to piggyback a token on the initial request from a client. This would eliminate the need to do a separate connection and would thereby reduce the impact made by Trinity compared to the TLS firmware, making them practically identical performance-wise. Figure 7.1 demonstrates the connection procedure where the token is piggybacked on the request and it is clear when comparing it to figure 5.2 that this reduces a large amount of overhead from the different connection setups.

[1]This could happen if the controller heap is corrupted or if the token has expired or been revoked

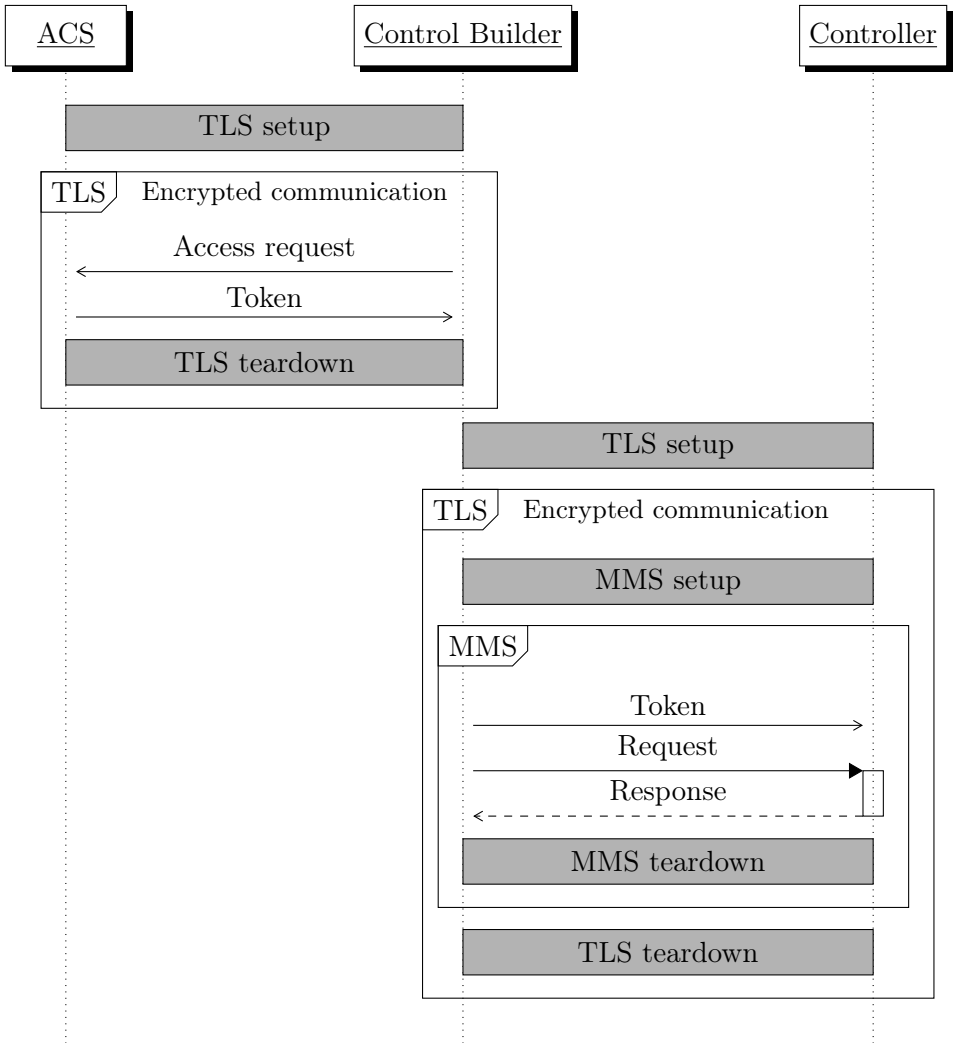

Figure 7.1: Sequence diagram of communication in Trinity with a piggybacked token

What we see from the results gathered in this test is that the increase in workload from the original firmware to a firmware with TLS protocol is quite large. However, should this protocol be implemented, the workload added by Trinity is negligible under normal working conditions and we therefore see no disadvantage of also adding access control with Trinity to the system. These results are in line with the findings in section 7.1.1, where we saw that the firmware size increased a lot when including the OpenSSL library, however the increase when adding Trinity on top of this was very small.

Chapter 8

Conclusions

8.1 Summary

The work in this thesis has been about implementing an access control system suitable for systems with constrained resources. The theory behind the access control system is presented in [2] and the goal of this thesis has been to implement the key aspects of this system in a real working environment, in this case an ABB control system. By measuring the performance of this system it was shown that implementing this access control scheme on top of client authentication, a prerequisite for an access control system, gives a negligible increase in resource consumption, successfully proving our initial thesis. This means that if you are going to add authentication to a system, you might as well also add access control in the form of Trinity.

Working on this master thesis has been interesting, challenging and rewarding. The biggest challenge was by far to get acquainted with the source code of the ABB system and to figure out how to implement Trinity into this existing framework. Design changes had to be made to accommodate this, however we have still been able to implement the core functionality of the original design as proposed in [2]. As we designed the system ourselves we also had to consider testing, both what was to be tested and how these tests would be performed. This was a new experience for both of us and is something we learned a lot by doing. Unfortunately we were not able to get as large a sample size as we initially wanted due to a memory leak in the TLS implementation, causing the heap to overflow after a number of TLS connections. We still feel however that we are able to show tendencies from which we are able to draw a fair conclusion. We are very pleased with both how the work has progressed and the final outcome of this thesis and we hope that the results gained can be used for future work in the field.

8.1.1 Contributions

All the design and implementation choices detailed below are original ideas that the authors have contributed to this thesis.

- A token was designed by us that would suit the ABB environment with the following properties:
  - It uses the distinguished name from X.509 certificates as subject identifier.
  - It gives access to multiple resources.
  - It is sent when a connection is first established between the controller and the CB and not when the protected resource is accessed.
  - It is sent using MMS variables.
- Token handling for the controller and the Control Builder was designed and implemented by us. This also includes adding an additional TLS stack to the CB for communication with the ACS.
- We chose to store the token as a specially designed "Hash entry", containing only the most necessary values, in hash tables. We also chose to store the token not only on the controller but also on the Control Builder in order to avoid unecessary token transmissions.
- We designed the interface for token transmission between the ACS and the CB.
- Error handling for Trinity was integrated into the existing error handling functionality of the controller and the CB.
- We decided which resources on the control system were best suited for access protection.
- We decided on using TLS for authentication and HMAC for token integrity.

8.2 Suggestions of Future Work for ABB

Since ABB has included authentication in their control system we would like to suggest, based on the results of this thesis, that ABB also implements Trinity. Doing so will give the system the extra functionality of authorization at basically no performance cost at all. We also feel that implementing Trinity to a production version of the control system would be beneficial for ABB as it would present opportunities for new use cases that would make the 800xA system more attractive to customers. Consider the scenario where the 800xA is used to control an oil rig in the middle of the North Sea. Should some error occur or system parameters need to be changed this might require the work of a special system engineer. This engineer is normally not stationed on the platform itself but would rather need to be flown in by helicopter, a time consuming and expensive process. Had it instead been possible to connect the control system to the Internet the maintenance could have been done by an engineer from his office on the mainland. This of course

presents problems regarding network security and this is where we feel Trinity could be used in order to only give access to the system engineer and stop all other attempted attacks.

Given below are some of the steps we feel need to be taken by ABB in order to implement a fully functioning version of Trinity to the control system.

- Improve on the structure of the code that implements the functionality of Trinity in order to make it more secure and more optimized, both on the controller and the Control Builder.
- Improve on the structure of the TLS implementation in order to make it more optimized.
- Investigate possible locations for the Access Control Server. There may exist suitable network locations already present in the framework to which this functionality could be integrated.
- Investigate different implementations of Access Control Servers. We do not recommend using the version written in this thesis for anything other than demonstration purposes.
- Design a structure for certificate distribution and access handling. This includes appointing a Certificate Authority and someone who maintains the ACS and decides upon an access hierarchy.
- Design and implement the functionality of token revocation, a feature currently not present in the system but one that is needed in order for the system to work in a real environment.

8.3 Related Work

The following section details related work that has been done in the field, but differ from the work done in this thesis in some key aspects.

8.3.1 Kerberos

Kerberos is an authentication protocol that provides a secure way of authenticating nodes over a non-secure network by use of a trusted third party. A brief summary of the protocol will be presented here, for the full specification we refer to [19].

When a client wishes to access a server it sends a request to an Authentication Server (AS) in plaintext. The request only consists of the client ID and when the AS receives this it searches its own database for the password matching this client ID. If found, a response is sent to the client consisting of a Client/TGS Session Key encrypted with the password found by the AS and a Ticket-Granting Ticket encrypted with the secret key of the TGS. TGS in this case stands for Ticket Granting Service, which may or may not be on the same machine as the AS.

The client decrypts the message containing the Client/TGS Session Key and sends a request containing the ID of the requested service to the TGS encrypted with this key along with the still encrypted Ticket-Granting Ticket. The TGS responds with a Client-to-Server ticket, including among others the Client-Server Session key, encrypted with the secret key of the requested service and a Client-Server Session Key, encrypted using the Client-TGS Session Key.

The encrypted Client-to-Server ticket is sent to the requested server along with an identifier of the requested service on the server, encrypted using the Client-Server Session Key. The server is able to decrypt these messages and the requested service is performed.

Differences to Trinity

Like Trinity, Kerberos uses a trusted third-party. The biggest difference is that in Trinity the third-party performs authorization, whereas in Kerberos it performs authentication. In Kerberos a client is given access to a server through a ticket, much like the tokens proposed in our system. However, upon receiving the ticket the client can request any of the possible services on the server and it is then, if necessary, up to the server itself to perform authorization of the client. As Trinity is designed to work in environments where the server might not have the resources to perform this type of authorization, this responsibility is instead given to the trusted third-party, the ACS.

A big drawback when using Kerberos in a constrained environment is that it is designed to incorporate message encryption through the use of symmetric keys for its protocol messages. In an environment where message encryption is not necessary this induces a lot of overhead to the communication, which is why Trinity does not in itself provide message encryption but instead relies on secure transport protocols to provide this service should it be needed. Should message encryption be left out altogether, Trinity will still be able to provide message integrity.

8.3.2 OAuth

OAuth is a protocol that provides a way for a client to access resources on behalf of a resource owner without the resource owner having to share its login credentials with the client. This is done by using a trusted third party, an authorization server, that issues access tokens that the client then uses to authenticate itself to the resource server and authorize access to the resources. A brief example of the usage of the protocol will be presented here, for a more thorough description of the protocol we refer to [20].

A user, resource owner, wants to use a mobile application, client, to access her photos that are stored on a server. The application requests access to these photos from the user, i.e. through a pop-up window that the user has to approve.

The application then requests access from the authorization server (AuthZ) us-

ing the access grant from the user as a parameter. The AuthZ responds with an access token containing the lifetime and scope of the access. This token is then used by the application when requesting access to the photos from the server. By using this protocol the user never has to give the application access to its full user credentials such as usernames and passwords, very useful in the internet today, where web pages often use logins of other services to provide their own services, e.g. news pages where you can log in to your Twitter account to comment on the news.

Differences to Trinity

The main difference between OAuth and Trinity is that in OAuth the user is actually the owner of the desired resources, she has simply outsourced her rights to grant access to these resources to a trusted third party. The AuthZ does not perform any authorization on the requesting client, it simply authenticates the client and then validates the request parameters to check if the request has been authorized by the resource owner, the user. The authorization is thereby done by the user, not the third party. In Trinity, neither the user nor the client is the owner of the desired resource and the trusted third party, the ACS, performs both authentication and authorization of the client to see if it has access to the desired resource.

A drawback of OAuth is that it does not provide message integrity on its tokens but instead relies heavily on secure transport protocols to avoid token interception. As mentioned above, Trinity applies the HMAC scheme to provide message integrity in such a way that should a token be intercepted by a malicious user, no information required to generate a new valid token can be extracted from this one. Message integrity for OAuth has been proposed in [21] that like Trinity involve adding a HMAC to the token.

8.4 Future Work

8.4.1 HMAC

HMAC was used in this thesis as a way of calculating MACs, mainly because of the built-in support in OpenSSL. However, there are a number of other schemes that might be more suitable for a real-world application. HMAC uses a relatively demanding hash-function in its calculations, something that might not be prefered in cases where the hardware in question is even more limited than the one used for this thesis. Such more optimized schemes include UMAC and Poly1305 [22]. By using one of these it is possible to decrease the load put on the hardware, however none of these have been tested. The use of HMAC is however justified by the fact that more and more constrained systems today feature dedicated cryptographic co-processors with support for SHA hashing. In order to support interoperability a SHA based HMAC is then still a valid choice. In this thesis SHA-1 hashing is used. For future implementations it is recommended to use stronger hash algorithms such as SHA-256, however the OpenSSL library used here did not support this newer version.

8.4.2 Clock Synchronization

As all access given to users in Trinity is time limited, it is very important that the internal clocks of all involved parties are synchronized. As this is not investigated in this thesis this is something that has to be considered in future implementations. The action of setting the internal clock of the Service Provider also provides the ability to manipulate the access control system as you are able to set the clock to a time where previously invalid tokens and X.509 certificates are now valid. This action thereby has to be protected in order to prevent these ”attacks” and a possible protection is Trinity itself, a valid token is needed in order to set the clock of an SP. Possible entities that would be given this access include the Access Control Server, as a trusted relationship is already established between it and the SP. This action should preferably be done periodically as often as needed in order to keep all clocks as synchronized as possible. This however presents the problem of initial clock synchronization, where it is possible that tokens and certificates might not yet have been distributed. A solution to this problem would be to have a trusted authority manually set the clock to the current time upon initially distributing certificates and shared secrets to the system and only after that initiate the periodical clock update from the ACS. This is considered a viable solution as most systems require some form of manual bootstrapping anyway for other purposes.

8.4.3 Token Formats

In Trinity the token consists of a string that is parsed by the controller upon validation. As mentioned before in this report this design choice is primarily based on the fact that the controller contains a string library and a way of transmitting string values easily in its original configuration. A different solution would have been to interpret the token as a bit value and use bit shifts of fixed lengths and other bit operations to compare values and thereby validate the token. Using string functions to compare values is quite heavy compared to simple bit operations, which is why this solution might be more cost effective than the one implemented in this thesis. The string library might also not always be implemented in systems with constrained resources as it might prove too large, another reason as to why bit operations might be preferred. A token might then have the characteristics shown in figure 8.1.

8.4.4 Token Storage

The tokens are currently stored by both the controller and the Control Builder. However, none of them make any effort to store the token should the system be turned off, meaning that in case of power outage for the controller and program termination for the CB, all stored tokens are lost. This is not an ideal scenario for any of them but it is especially bad for the CB. Being a Windows application chances are that it will be turned on and off several times each day, thereby increasing the number of unnecessary token transmissions to the controllers due to the fact that the CB no longer remembers which tokens have already been sent. On the controller the problem is not as prominent as this machine is designed and

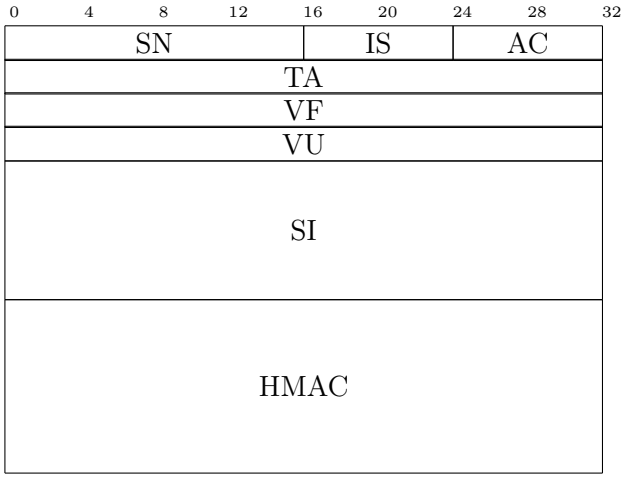

Figure 8.1: Example of a token in bit format

built to be run for very long periods of time without being turned off. A solution to this problem would be to copy the stored tokens to a non-volatile memory location when the system is turned off. In the CB this could mean adding code to the shutdown routine of the program that copies the token information to a file stored in a non disclosed folder, a file that is read upon every start-up. The same would apply for the controller, but here code would also need to be added to the emergency routines as a controller might suffer from unplanned power outages.

8.4.5 Token Revocation

This implementation of Trinity does not include token revocation, however there exist the possibility of expanding the system to also include this procedure. In a real implementation this functionality is a necessity, as access policies may change over time. One way of doing token revocation would be to have the ACS send the Sequence Numbers of the tokens being revoked to all connected controllers. These controllers could then parse through all their stored tokens until a match is found and then remove this token from the system. The Sequence Number is already present in the token, the changes needed in order to implement the revocation procedure lie elsewhere. First of all a structure of how the revocation should be done has to be designed, meaning when and how revocations are to be done and by whom. This functionality then needs to be added to the ACS and the controller. Changes also has to be made to the Control Builder in order to deal with error messages from the ACS regarding revocation.

Bibliography

[1] F. Mattern and C. Floerkemeier. From the internet of computers to the internet of things. Technical report, Distributed Systems Group, Institute for Pervasive Computing, ETH Zurich. URL http://www.vs.inf.ethz.ch/publ/papers/Internet-of-things.pdf.

[2] G. Selander, M. Sethi, and L. Seitz. Access Control Framework for Constrained Environments. Internet-Draft, CoRE Working Group, February 2014. URL https://tools.ietf.org/html/draft-selander-core-access-control-02. Work In Progress.

[3] About SICS Swedish ICT. URL https://www.sics.se/about-sics. Last checked: 2014-04-23.

[4] Who we are - ABB in brief. URL http://new.abb.com/about/abb-in-brief. Last checked: 2014-04-23.

[5] D. Ince. *Message Authentication Code.* Oxford University Press. ISBN 9780199571444.

[6] H. Krawczyk, M. Bellare, and R. Canetti. HMAC: Keyed-Hashing for Message Authentication. Request For Comments (RFC) 2104, Internet Engineering Task Force (IETF), February 1997. URL http://www.ietf.org/rfc/rfc2104.txt.

[7] T. Krovetz. UMAC: Message Authentication Code using Universal Hashing. Request For Comments (RFC) 4418, Internet Engineering Task Force (IETF), March 2006. URL http://www.ietf.org/rfc/rfc4418.txt.

[8] P. Gallagher. Digital Signature Standard (DSS). Federal Information Processing Standards Publication (FIPS) 186-4, National Institute of Standards and Technology (NIST), July 2013. URL http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.186-4.pdf. p. 9.

[9] D. Eastlake and P. Jones. US Secure Hash Algorithm 1 (SHA1). Request For Comments (RFC) 3174, Internet Engineering Task Force (IETF), September 2001. URL http://www.ietf.org/rfc/rfc3174.txt.

[10] R. Rivest. The MD5 Message-Digest Algorithm. Request For Comments (RFC) 1321, Internet Engineering Task Force (IETF), April 1992. URL http://www.ietf.org/rfc/rfc1321.txt.

[11] T. Dierks and E. Rescorla. The Transport Layer Security (TLS)Protocol Version 1.2. Request For Comments (RFC) 5246, Internet Engineering Task Force (IETF), August 2008. URL http://www.ietf.org/rfc/rfc5246.txt. p. 3-4, 14-15, 25-26.

[12] Information Sciences Institute University of Southern California. Transmission Control Protocol (TCP). Request For Comments (RFC) 793, Internet Engineering Task Force (IETF), September 1981. URL http://www.ietf.org/rfc/rfc793.txt.

[13] S. Farrell S. Boeyen R. Housley W. Polk D. Cooper, S. Santesson. Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile. Request For Comments (RFC) 5280, Internet Engineering Task Force (IETF), May 2008. URL http://tools.ietf.org/rfc/rfc5280.txt.

[14] SISCO, INC. Overview and Introduction to the Manufacturing Message Specification (MMS). Technical report, August 1995. URL http://www.sisconet.com/downloads/mmsovrlg.pdf. p. 1-3.

[15] J. Loughney G. Zorn V. Fajardo, J. Arkko. Diamater Base Protocol. Request For Comments (RFC) 6733, Internet Engineering Task Force (IETF), October 2012. URL http://tools.ietf.org/html/rfc6733.

[16] IBM, Corp. Security Server (RACF) Introduction. Technical report, September 1999. URL http://publibz.boulder.ibm.com/epubs/pdf/ich1a510.pdf.

[17] R. Engelschall. Openssl: The open source toolkit for ssl/tls. URL http://www.openssl.org/about/. Last checked: 2014-03-28.

[18] M. McDowell. Understanding Denial-of-Service Attacks. URL http://www.us-cert.gov/ncas/tips/ST04-015. Last checked: 2014-04-23.

[19] C. Neuman, T. Yu, S. Hartman, and K. Raeburn. The Kerberos Network Authentication Service (V5). Request For Comments (RFC) 4120, Internet Engineering Task Force (IETF), July 2005. URL http://www.ietf.org/rfc/rfc4120.txt.

[20] D. Hart. The OAuth 2.0 Authorization Framework. Request For Comments (RFC) 6749, Internet Engineering Task Force (IETF), October 2012. URL http://www.ietf.org/rfc/rfc6749.txt.

[21] M. Jones and D. Hart. The OAuth 2.0 Authorization Framework: Bearer Token Usage. Request For Comments (RFC) 6750, Internet Engineering Task Force (IETF), October 2012. URL http://www.ietf.org/rfc/rfc6750.txt.

[22] D. Bernstein. Poly1304-aes: a state-of-the-art message-authentication code. URL `http://cr.yp.to/mac.html`. Last checked: 2014-03-28.

Appendix A
Division of work

The following sections have been written jointly:

- Chapter 7 (Discussion)
- Chapter 8 (Conclusions)

The following sections have been written by Niklas:

- Section 2.1 (Message Authentication Code)
- Section 2.4 (Manufacturing Message Specification)
- Chapter 3 (Proposed Framework)
- Section 4.3 (Security Considerations)
- Section 4.4 (Summary)
- Section 5.2 (Communication between Involved Parties)
- Section 5.3 (Token)
- Section 5.6 (Service Provider - Controller)
- Section 8.3 (Related Work)
- Section 8.4 (Future Work)

The following sections have been written by Jonas:

- Chapter 1 (Introduction)
- Section 2.2 (Transport Layer Security)
- Section 2.3 (Public Key Certificate)
- Section 4.1 (Hardware)
- Section 4.2 (Software)
- Section 5.1 (Delimitations)
- Section 5.4 (Access Control Server)
- Section 5.5 (Client - Control Builder)
- Chapter 6 (Evaluation of the Implementation)

Appendix B
Test Results

Table B.1: Trinity firmware - Connection duration during *Show Downloaded Items*

Connection duration (ms)		
Controller 1	Controller 2	Controller 3
1245.93	1275.55	1322.29
1225.86	1263.61	1267.53
1224.19	1280.26	1282.36
1213.4	1277.32	1266.3
1122.84	1273.22	1271.77
1222.91	1230.27	1284.45
1223.89	1280.3	1276.33
1235.86	1269.17	1277.01
1224.99	1268.92	1271.98
1274.04	1290.58	1276.02
1225.09	1277.73	1293.69
1253.85	1294.11	1269.38
1224.47	1277.3	1334.83
1200.68	1293.07	1277.19
1262.69	1278.86	1272.28
1244.35	1283.03	1272.9
1233.81	1317.43	1323.43
1223.55	1285.14	1273.43
1223.45	1372.87	1260.76
1224.27	1273.63	1326.39

Table B.2: Trinity firmware - Token connection duration during *Show Downloaded Items*

Connection duration (ms)		
Controller 1	Controller 2	Controller 3
1235.36	1397.01	1326.52
1216.15	1337.64	1326.41
1265.75	1392.16	1316.51
1275.82	1345.24	1304.12
1236.02	1382.69	1326.22
1213.42	1344.81	1306.57
1234.57	1396.03	1337.25
1215.32	1364.77	1345.38
1194.88	1396.23	1364.59
1266.36	1385.85	1315.99
1270.38	1476.56	1325.64
1276.52	1370.6	1376.76
1206.66	1377.49	1345.15
1203.05	1395.88	1337.63
1286.18	1386.37	1374.96
1288.8	1397.46	1326.49
1285.92	1407.23	1336.03
1246.43	1393.06	1305.57
1215.29	1396.55	1320.65
1235.66	1319.28	1305.96

Table B.3: Trinity firmware - Connection duration during *Show MMS Variables*

Connection duration (ms)		
Controller 1	Controller 2	Controller 3
634.242	651.259	636.623
574.189	651.71	615.356
565.922	645.139	613.3871
564.061	639.134	600.503
567.069	646.761	605.592
545.647	645.44	596.422
575.334	648.681	604.755
576.42	636.617	615.184
577.157	642.527	630.368
596.267	643.579	652.261

Table B.4: Trinity firmware - Token connection duration during *Show MMS Variables*

Connection duration (ms)		
Controller 1	Controller 2	Controller 3
1015.37	1174.61	1097.2
1013.19	1168.52	1095.48
1045.52	1159.62	1114.74
1025.99	1183.1	1114.85
1014.23	1135.87	1117.85
1015.19	1145.66	1137.73
1012.41	1239.74	1113.48
1085.62	1145.39	1196.99
1035.09	1155.73	1103.78
1015.78	1123.82	1141.44

Table B.5: Trinity firmware - Connection duration during *Show MMS Connections*

Connection duration (ms)		
Controller 1	Controller 2	Controller 3
585.711	642.956	615.603
573.556	628.709	633.321
554.792	626.593	639.887
565.482	656.986	629.397
575.546	635.203	615.803
575.584	629.707	626.448
585.296	636.535	605.566
603.167	631.589	615.983
596.233	639.335	607.555
566.764	639.76	634.876

Table B.6: Trinity firmware - Token connection duration during *Show MMS Connections*

Connection duration (ms)		
Controller 1	Controller 2	Controller 3
1025.69	1144.49	1124.94
1025.29	1141.33	1111.18
1044.84	1148.35	1134.56
1033.22	1154.21	1121.11
1034.23	1153.81	1105.34
1013.42	1150.81	1159.17
1034.58	1141.07	1124.74
1025.3	1158.14	1122.36
1036.62	1164.93	1120.83
1027.79	1156.99	1126.18

Table B.7: Trinity firmware - Connection duration during *Show Firmware Information*

Connection duration (ms)		
Controller 1	Controller 2	Controller 3
1025.53	1090.43	1096.26
1045.9	1045.15	1096.09
1043.81	1095.98	1103.72
993.818	1138.06	1095.44
1043.59	1093.73	1085.62
1043.39	1064.92	1056.39
993.264	1098.81	1094.59
992.879	1085.51	1043.9
993.151	1098.47	1095.42
993.053	1094.26	1093.95

Table B.8: Trinity firmware - Token connection duration during *Show Firmware Information*

Connection duration (ms)		
Controller 1	Controller 2	Controller 3
1273.7	1102.34	1343.12
1225.57	1122.67	1337.68
1216.86	1122.22	1315.55
1296.5	1168.8	1434.28
1226.01	1121.82	1315.67
1266.2	1122.79	1336.43
1245.86	1072.43	1355.64
1263.94	1068.72	1325.52
1236	1074.24	1325.83
1217.01	1122.32	1325.37

Table B.9: TLS firmware - Connection duration during *Show Downloaded Items*

Connection duration (ms)		
Controller 1	Controller 2	Controller 3
1232.12	1274.21	1287.18
1205.34	1257.19	1287.09
1217.06	1324.03	1270.34
1225.47	1262.78	1278.78
1216.03	1266.74	1270.51
1226.36	1276.26	1280.94
1228.25	1331.45	1269.59
1191.44	1209.79	1271.11
1225.09	1271.87	1268.36
1234.43	1274.64	1265.24
1224.97	1318.64	1270.39
1224.85	1282.35	1273
1225.6	1274.55	1268.41
1226.19	1319.74	1284.19
1227.71	1312.16	1274.81
1234.6	1276.83	1274.74
1225.78	1273.08	1275.67
1233.28	1274.82	1268.54
1226.04	1321.78	1262.93
1234.83	1326.63	1270.47

Table B.10: TLS firmware - Connection duration during *Show MMS Variables*

Connection duration (ms)		
Controller 1	Controller 2	Controller 3
585.641	662.487	616.348
573.932	695.367	695.294
575.044	631.565	612.906
574.986	628.486	638.552
595.787	629.906	609.65
575.141	625.141	624.985
572.691	620.894	638.528
599.864	666.875	641.591
585.238	664.973	634.505
576.402	665.222	631.072

Table B.11: TLS firmware - Connection duration during *Show MMS Connections*

Connection duration (ms)		
Controller 1	Controller 2	Controller 3
575.543	623.404	626.986
636.048	645.729	605.625
577.004	655.137	639.236
585.474	628.921	638.373
565.859	660.485	636.296
582.442	647.188	635.955
578.357	644.66	637.403
578.757	665.395	629.009
566.706	638.271	636.074
575.722	651.086	615.878

Table B.12: TLS firmware - Connection duration during *Show Firmware Information*

Connection duration (ms)		
Controller 1	Controller 2	Controller 3
945.721	1098.58	1096.8
1045.22	1104.11	1085.81
995.333	1093.09	1094.07
995.154	1109.11	1054.68
1045.2	1105.6	1088.03
1045.06	1096.04	1103.62
1044.93	1105.15	1096.78
995.407	1094.18	1086.64
1045.19	1040.86	1081.91
995.535	1043.45	1056.31

Table B.13: Original firmware - Connection duration during *Show Downloaded Items*

Connection duration (ms)		
Controller 1	Controller 2	Controller 3
770.456	781.062	751.639
760.058	790.838	801.602
771.541	770.878	772.438
751.494	761.135	803.431
771.165	750.579	770.775
750.736	741.11	749.237
770.398	773.065	761.527
790.087	753.109	802.971
770.464	770.829	770.752
750.639	750.895	771.853
770.044	770.999	770.734
750.713	800.865	751.553
760.581	760.421	771.381
750.116	760.782	790.793
769.116	761.024	780
760.133	751.651	756.757
768.795	765.371	771.079
750.179	801.121	751.388
771.427	750.575	770.609
783.426	740.965	751.248

Table B.14: Original firmware - Connection duration during *Show MMS Variables*

Connection duration (ms)		
Controller 1	Controller 2	Controller 3
80.107	81.305	81.324
81.14	81.315	81.216
80.955	83.629	79.863
81.099	81.421	81.224
81.161	90.966	81.278
81.108	81.233	83.501
81.896	81.497	81.209
81.206	80.359	82.037
81.23	80.297	81.067
80.423	81.581	80.912

Table B.15: Original firmware - Connection duration during *Show MMS Connections*

Connection duration (ms)		
Controller 1	Controller 2	Controller 3
81.14	80.473	84.08
70.528	80.944	81.201
81.06	81.017	81.869
81.146	78.286	81.274
81.177	80.341	80.813
81.224	70.285	70.97
101.33	79.346	81.072
80.946	102.492	82.099
81.66	82.014	81.273
81.008	81.155	81.147

Table B.16: Original firmware - Connection duration during *Show Firmware Information*

Connection duration (ms)		
Controller 1	Controller 2	Controller 3
530.987	521.846	541.647
545.992	541.186	590.526
540.755	590.944	541.647
540.056	541.116	541.657
539.651	590.993	541.604
540.852	528.333	543.147
529.579	541.052	592.15
541.106	591.512	541.637
540.597	590.93	541.763
531.575	591.21	512.785

LUND
UNIVERSITY

Series of Master's theses
Department of Electrical and Information Technology
LU/LTH-EIT 2014-409

http://www.eit.lth.se