

Master's Thesis

Authorization Aspects of the Distributed Dataflow-oriented IoT Framework Calvin

Tomas Nilsson



Authorization Aspects of the Distributed Dataflow-oriented IoT Framework Calvin

Tomas Nilsson
`elt11tni@student.lu.se`

Department of Electrical and Information Technology
Lund University

Advisors:
Martin Hell, Lund University
Håkan Englund, Ericsson Research

June 8, 2016

Printed in Sweden
E-huset, Lund, 2016

Abstract

The evolution into a networked society, where a wide variety of devices are connected to the Internet, opens up many new opportunities for creating smart products and applications that dynamically adapt to the current environment. A common scenario will be that applications require or benefit from using several devices at the same time for the execution. Such distributed applications may be complex to write for an application developer. The goal of the open-source framework Calvin, developed by Ericsson Research, is to make it possible for developers of distributed applications to focus on their ideas instead of the complex implementation details.

A user may specify some details about where the application is allowed to execute. Based on this information and other parameters, Calvin will automatically decide where it is most beneficial for different parts of the application to execute. Calvin can also handle migration to other devices without interrupting the execution of the application.

This dynamic distributed execution model results in challenges when it comes to deciding what resources specific applications, running on behalf of different users, should be allowed to access on a specific device. In this thesis, an authorization framework, based on fine-grained attribute-based access control, is proposed as a solution for the access control in Calvin. Flexibility and compact message formats are some of the most important aspects of the design in order to support different devices with different constraints. The proposed authorization solution has been implemented and is now available as a part of the Calvin framework.

Keywords: Authorization, Attribute-Based Access Control, Calvin, Distributed Computing, Internet of Things

Acknowledgements

This report is my master's thesis for a degree in Electrical Engineering from the Faculty of Engineering at Lund University. The project was carried out at Ericsson Research in Lund. I would like to thank the people that helped and supported me during my work on this thesis.

First of all, I would like to express my gratitude to the members of the Platform Security group at Ericsson Research in Lund for always being nice and helpful. It has been a pleasure to be a part of your group while working on this thesis. I would especially like to thank Håkan Englund, my advisor at Ericsson, for interesting discussions and guidance throughout my time at Ericsson.

Thanks also to Harald Gustafsson and the other Calvin developers in the Cloud Technology group at Ericsson Research in Lund for being helpful when I have had questions about Calvin.

I would also like to thank Martin Hell, my advisor at Lund University. The enthusiasm Martin shows when teaching security courses and the knowledge you get from his lectures have inspired me to do my master's thesis within security, and he has been helpful when I have asked for advice during the work on this thesis.

Table of Contents

1	Introduction	1
1.1	Aims and Challenges	1
1.2	Related Work	2
1.3	Thesis Outline	3
2	Theory	5
2.1	Authentication, Authorization, and Access Control	5
2.2	Access Control Models	5
2.2.1	Discretionary Access Control	6
2.2.2	Mandatory Access Control	6
2.2.3	Role-Based Access Control	6
2.2.4	Attribute-Based Access Control	6
2.3	XACML	7
2.3.1	Reference Architecture	7
2.3.2	Request/Response Language	8
2.3.3	Policy Language	8
2.3.4	SAML Profile	9
2.3.5	JSON Profile	9
2.3.6	REST Profile	9
2.4	Asymmetric Cryptography	10
2.4.1	Digital Signatures	10
2.4.2	Public Key Infrastructure and Certificates	10
2.5	JSON Web Token	10
2.5.1	Header	11
2.5.2	Payload	12
2.5.3	Signature	12
3	Calvin – Merging Cloud and IoT	15
3.1	Distributed Cloud for IoT	15
3.2	Applications and Actors	15
3.3	Runtime	17
3.4	Migration, Capabilities, and Requirements	17
3.5	Security	19
3.6	Example	19

4	Designing and Implementing Authorization in Calvin	21
4.1	Attribute-Based Access Control	21
4.1.1	Subject Attributes	23
4.1.2	Resource Attributes	23
4.1.3	Action Attributes	24
4.1.4	Environment Attributes	24
4.2	Policy Enforcement Point (PEP)	24
4.2.1	Runtime Registration	26
4.2.2	Authorization Requests/Responses	26
4.2.3	JWT for External Authorization	26
4.3	Policy Decision Point (PDP)	27
4.3.1	Find Matching Policies	28
4.3.2	Evaluate Policies	28
4.3.3	Combine Policy Decisions	30
4.4	Policy Information Point (PIP)	31
4.5	Policy Retrieval Point (PRP)	31
4.6	Policy Administration Point (PAP)	31
4.7	Smart Migration	32
5	Authorization Tests in Calvin	35
5.1	Correctness Tests	35
5.2	Performance Tests	37
6	Discussion	39
6.1	Comments on Test Results	39
6.2	Design Choices	40
6.2.1	JSON-based XACML	40
6.2.2	Adaptable to Constrained Devices	40
6.3	Security Considerations	41
6.4	Privacy Considerations	41
6.5	Future Work	42
7	Conclusion	43
	References	45
A	Runtime Registration Example	47
B	Authorization Request/Response Example	49
B.1	Request	49
B.2	Response	50
C	Authorization Policy Example	51

List of Figures

2.1	The structure of a JSON Web Token	11
3.1	Actor model in Calvin	16
3.2	Calvin migration at deployment time based on requirements/capabilities matching	18
3.3	A Calvin script and dataflow graph example	20
4.1	New authorization flow implemented in Calvin	22
4.2	New deployment process in Calvin	25
4.3	Smart migration when access is denied.	33

List of Tables

4.1	Functions that can be used in a condition in a policy	29
4.2	REST API for policy management	32
5.1	Correctness tests and results for Calvin authorization	35
5.2	Performance test results for Calvin authorization	37

Abbreviations

ABAC	Attribute-Based Access Control
API	Application Programming Interface
CA	Certificate Authority
DAC	Discretionary Access Control
DHT	Distributed Hash Table
ECC	Elliptic Curve Cryptography
ECDSA	Elliptic Curve Digital Signature Algorithm
IoT	Internet of Things
JSON	JavaScript Object Notation
JWT	JSON Web Token
MAC	Mandatory Access Control <i>or</i> Message Authentication Code
PAP	Policy Administration Point
PDP	Policy Decision Point
PEP	Policy Enforcement Point
PIP	Policy Information Point
PKI	Public Key Infrastructure
PRP	Policy Retrieval Point
RBAC	Role-Based Access Control
REST	Representational State Transfer

SAML	Security Assertion Markup Language
TLS	Transport Layer Security
XACML	eXtensible Access Control Markup Language
XML	Extensible Markup Language

The number of devices that are connected to the Internet is growing at a rapid pace, creating what is commonly referred to as the *Internet of Things* (IoT) [1]. By putting sensors on a wide variety of things and letting them communicate with other devices, it is possible to create smart products and services that adapt to the current environment. This transformation into a networked society, where everything that can benefit from a connection will be connected, is expected to largely influence everyday life.

Many tiny devices are constrained in storage or processing power. Hence, there is a need to offload certain tasks to external computing resources. *Cloud computing* is a frequently used term that refers to shared computing resources that provide services over the Internet [2]. Using the cloud for computation offloading or storage is getting more and more popular.

The open-source framework *Calvin* [3] is developed by Ericsson Research to simplify the development of distributed applications combining IoT and cloud computing. An application developer should not need to worry about different communication protocols and other details. Calvin uses a dataflow programming methodology with actors that perform certain tasks. An application has a description of which actors to use, how data flows between the actors and some requirements on the runtime environment.

When an application starts it is not always decided on which physical devices the application will execute. Instead, different parts of the application may execute where it is most beneficial. An actor may even be migrated to another runtime without interrupting the execution. This thesis focuses on the security challenge of handling access control/authorization for different runtimes in this dynamic environment. It must be possible to decide if execution of an actor or application deployed by a certain user should be allowed right now on a specific runtime.

1.1 Aims and Challenges

The following aims were set for this master's thesis work:

- Research the best approach for authorization of applications/actors in the Calvin framework.
- Implement the findings of the theoretical work into Calvin.

A number of desired features and requirements on the authorization solution were determined before starting the research:

- Fine-grained authorization decisions on access to resources offered by a runtime are required. It must be possible to make decisions based on many different parameters.
- A solution that is adaptable to different environments is desirable.
- The authorization must be designed in such a way that it can be used as input for migration decisions in Calvin.

The main challenge is that Calvin has a distributed execution model, where parts of the application are executed on different runtimes not known when the execution of the application starts. Using a standard approach, where the user gets a token that is forwarded to the runtime when access to a resource is required, is difficult in this case since there must be a token for every possible runtime that may execute the application. Since those runtimes are not known at deployment time, an alternative approach is needed for Calvin.

Another challenge is that the solution must be suitable for different IoT situations in constrained environments. Network nodes may be limited in storage or processing power, and devices may for example not be able to handle authorization themselves.

1.2 Related Work

The ACE (*Authentication and Authorization for Constrained Environments*) working group within IETF (*Internet Engineering Task Force*) investigates how security standards need to be adapted to work for IoT devices with for example limited processing power [4]. Their documents include many useful authorization aspects, but do not discuss a dynamic distributed execution model like the one used by Calvin.

The working group has for example defined how to use the well-known OAuth 2.0 standard as an authorization framework for IoT [5]. OAuth is used to delegate access by issuing access tokens that are later presented when access to a protected resource is required. As mentioned earlier, such an authorization solution is not suitable for Calvin since it is not known in advance where execution will take place. However, inspiration has been taken from the usage of JSON Web Tokens to carry information in the proposed OAuth 2.0 IoT authorization framework.

Another authorization framework for IoT, based on the access control standard XACML, has been proposed by Seitz et al [6]. For the same reasons as in the OAuth case, this XACML-based framework is not a perfect match for the authorization in Calvin, but the use of XACML to enable fine-grained access control has been used as inspiration for this thesis. Different standards have to be combined in a new way to be suitable for Calvin.

1.3 Thesis Outline

Chapter 2 gives a brief introduction to theory that is needed to understand the thesis.

Chapter 3 describes what Calvin is and how it works.

Chapter 4 presents the design and implementation details of the authorization solution that is proposed in this thesis.

Chapter 5 contains test results for the authorization implementation.

Chapter 6 is dedicated to discussions about test results, design choices, security considerations, and future work.

Chapter 7 summarizes the main points of this thesis work.

Appendix A-C include examples to show how the authorization implementation can be used.

Many existing protocols and ideas for authorization and secure communication have been used as a basis for this work, either as inspiration or as a part of the implementation. This chapter gives a brief overview of theory that is needed to understand the following chapters in this thesis.

2.1 Authentication, Authorization, and Access Control

Authentication, *authorization*, and *access control* are three commonly used terms in computer security. The terms are often mixed up, and a common misunderstanding is that they have more or less the same meaning. However, there is a fundamental difference between authentication and authorization, whereas access control is a more general term [7].

Authentication is the process of verifying *who you are*. Typically a user enters a username and a password, and these user credentials are checked against the user information stored by the authentication system. As an alternative to passwords, authentication systems may for example use smart cards or fingerprints to verify that the user is who he/she claims to be.

Authorization is the process of determining *what you are allowed to do*. When an authenticated user tries to perform certain operations on privileged resources, the system uses some kind of rules to check if that user has permission to perform that operation. Different users typically have different permissions in a system. Some of the users could for example be administrators that are able to create, edit, and delete content, whereas other users only have read-access. Authorization is usually needed after a successful authentication to secure a system.

Access control is a general term that refers to all possible ways of controlling access to a resource. It typically involves the use of authentication and authorization, but the access could also be controlled only based on for example the time of the day.

2.2 Access Control Models

There are many different methodologies used to ensure that no unauthorized parties get access to protected resources. This section briefly introduces the most common access control models.

2.2.1 Discretionary Access Control

Discretionary Access Control (DAC) is a type of access control where the owner of the resource controls the access [8]. The owner decides who is allowed to access the resource and what type of operations (for example read, write, execute) different users have permission to do. DAC models are for example used in operating systems to allow the owner of a file to give access of different levels to other users.

2.2.2 Mandatory Access Control

Mandatory Access Control (MAC) refers to an access control model in which access to different resources is centrally controlled instead of being controlled by the owner of the resource [8]. It is usually used to protect highly sensitive information in for example government organizations. Each resource can be given a sensitivity label (e.g. "secret" or "top secret"), and similarly users have different labels specifying the level of sensitive information they have permission to access. For example, if a user tries to access a file classified as "top secret", access will be denied if the user only has permission to access resources classified as "secret".

2.2.3 Role-Based Access Control

In the *Role-Based Access Control* (RBAC) model, permission to access a resource is given to roles instead of individual users [8]. The concept of roles is common for other purposes in organizations and companies, which makes RBAC a natural choice in such scenarios. A user may have different roles in different contexts. Access decisions are based on the roles a user has been assigned in the system. Similar to MAC, the access is hence centrally controlled by the system instead of being controlled by the owner of the resource.

By controlling access based on roles instead of individual users, administration is simplified significantly when users for example join, leave, or change departments. RBAC is widely used in many large systems.

2.2.4 Attribute-Based Access Control

Attribute-Based Access Control (ABAC) is an access control model where access is granted or denied by evaluating policy rules against attributes which describe the entity requesting access, the resource for which access is being requested, and the current environment relevant for the request [9]. Attributes consist of a name and a value, which has some kind of information about the request.

The key difference between ABAC and RBAC is that ABAC decisions can be based on many different attributes instead of just the role attribute. By also using environment attributes, for example the time of the day or the current location, ABAC also makes the access decisions dynamic instead of having static rules.

The use of arbitrary attributes in ABAC enables a much more flexible and fine-grained access control than what is possible when RBAC is used [10]. There are many real-life scenarios where access must be controlled based on several different conditions. A common example is if a user needs access to some files owned by another department. Granting access for the user to all files belonging to that

department is usually not a desirable alternative. If RBAC is used, a new role that only has permission to access the desired files has to be created. In the same way, additional roles have to be created if other persons need access to other sets of files. The phenomenon where a large number of roles have to be created to enforce rules of finer granularity is called *role explosion*. Using ABAC, the problem is solved by adding another logical condition to a policy rule instead of adding more roles. This way of focusing on rules instead of roles resembles the actual business needs and is one of the reasons why ABAC increases in popularity.

2.3 XACML

XACML (*eXtensible Access Control Markup Language*) is an OASIS standard based on the concept of attribute-based access control (ABAC) [11, 12]. The standard defines both a policy language and a request/response language for authorization. In addition to these two parts, a reference architecture is also proposed in the standard. The XACML Technical Committee has also written several profiles that explain how to use XACML in different well-defined settings or extend the standard with new functionality.

2.3.1 Reference Architecture

The XACML standard specifies the architecture and process for evaluating authorization requests against policies and returning a response.

The standard uses the following names for the different entities involved in the XACML architecture [12, 13]¹:

- **Policy Enforcement Point (PEP)** – Intercepts users' requests to access resources by sending decision requests and enforcing the received authorization decisions.
- **Policy Decision Point (PDP)** – Evaluates decision requests against policies to make authorization decisions.
- **Policy Retrieval Point (PRP)**² – Stores the authorization policies (for example a database or the file system).
- **Policy Information Point (PIP)** – Acts as a source of attribute values to provide additional attributes that are not included in the decision request.
- **Policy Administration Point (PAP)** – Used to manage (create, update, remove) authorization policies.

When a user wants to perform a certain action on a protected resource, the PEP will intercept the request and send an authorization decision request to the PDP.

¹The communication between the different entities is illustrated in Figure 4.1.

²Policy Retrieval Points (PRP) are not defined in the latest version of the XACML standard, but they were mentioned in earlier versions and the term is still used in several implementations of XACML [13].

By using the PRP to look up policies and evaluate the request against these policies, the PDP is able to decide whether access should be permitted or not. If there are attributes in the policies that are not present in the request, the PDP will use a PIP to retrieve these attribute values. For example, if the request contains the username but the policy uses the role of the user, a PIP which has information about the roles associated with a user will be able to return the role attribute value to the PDP.

The policy evaluation leads to an authorization decision which is returned by the PDP to the PEP. Based on the decision in the response from the PDP, the PEP will either grant access or deny the request.

2.3.2 Request/Response Language

XML (*Extensible Markup Language*), which has a widespread support from large platforms and vendors, is used for the requests and responses in XACML [12].

A decision request is sent by a PEP to a PDP to get an authorization decision. The request contains attributes describing the subject, action, resource, and environment for the requested access.

When the attributes of the request have been compared to attribute values in the policies, the authorization decision (*Permit*, *Deny*, *NotApplicable*, or *Indeterminate*) is returned as a response from the PDP to the PEP. The response may also contain directives (*Obligations*) from the PDP to the PEP on additional operations that must be performed when enforcing the decision.

2.3.3 Policy Language

Just like the requests and responses, the XACML policies are also written in XML [12]. Policies are used to describe access control requirements. The XACML standard allows quite complex policies, but all details will not be explained here.

In order to get a basic understanding of XACML policies, some terminology used in the standard has to be explained:

- **Target** – A set of simple conditions for different attributes to determine to which requests a policy or a rule applies.
- **Rule** – Part of a policy which contains a *Target*, an *Effect*, a *Condition*, and, optionally, an *Obligation*.
- **Condition** – An advanced form of a *Target* which uses functions that evaluate the truth of statements about attributes.
- **Obligation** – An operation, specified in a policy, that should be performed by the PEP when enforcing an authorization decision.

When a request is received, the request attributes will be compared to the *Target* of each policy to determine against which policies the request should be evaluated. A policy contains one or many *Rules* that are used to render an authorization decision. If the *Condition* in the rule evaluates to true, the rule effect (*Permit*

or *Deny*) will be returned, possibly together with *Obligations*. Otherwise, the decision will be *Indeterminate* (if an error occurred) or *NotApplicable*.

The XACML standard specifies a number of combining algorithms that define how to combine the decisions of different rules into a single decision, for example *Deny Overrides* which means that the decision will be *Deny* if any of the rule evaluations return *Deny*. Combining algorithms may also be used to combine decisions from multiple policies into a final decision if multiple policies match the request.

2.3.4 SAML Profile

The *XACML SAML Profile* defines how to combine XACML and the *Security Assertion Markup Language* (SAML) 2.0 [14]. The XACML standard itself does not specify how assertions, protocols, and transport mechanisms between different entities are implemented. Instead, it has to be complemented by other standards for a full implementation of the XACML usage model. SAML is an OASIS standard that can be used for this purpose since it defines XML schemas for different types of requests and responses with security assertions.

By embedding XACML requests and responses in SAML assertions where the information is signed, it is possible for the recipient to verify that the message comes from the correct sender and that the information has not been changed.

2.3.5 JSON Profile

The *JSON Profile of XACML* defines a JSON (*JavaScript Object Notation*) format for the XACML request and response [15]. JSON is getting more and more popular as data exchange format thanks to its simplicity. An important aspect for the authors has been to make sure that it is possible to translate XACML requests and responses from the XML representation to the JSON representation, and the other way around, without losing any information. The representations must be equivalent.

Currently, the JSON profile only deals with the messages sent between the PEP and the PDP. Compared to XML, JSON is much more compact and easier to read. The idea of the profile is to remove the verbose aspects of XACML and provide a more lightweight alternative for the XACML requests and responses.

2.3.6 REST Profile

The *REST Profile of XACML* explains how to use XACML in a REST (*Representational State Transfer*) architecture, which means using a standardized architecture, based on HTTP standard methods, for referencing and manipulating resources in the interaction between a client and a server [16]. This is useful if an external PDP is used by several PEPs to get authorization decisions. The profile defines for example how the XACML request is sent from the PEP to the PDP as part of a HTTP POST request.

2.4 Asymmetric Cryptography

Asymmetric Cryptography, also called *Public-Key Cryptography*, uses a key pair consisting of a public key, which can be shared with others, and a private key, which is only known by the owner [17]. The keys are linked together mathematically and can be used to perform operations such as encryption, decryption, signature generation, and signature verification. A brief overview of asymmetric cryptography, with focus on digital signatures, will be provided here.

An important property of the keys is that it must be computationally infeasible to determine the private key given the public key and information about the cryptographic algorithm [17]. *RSA* is a commonly used public-key algorithm based on the mathematical difficulties of finding the prime factors of large numbers. Another way to realize asymmetric cryptography is to use *Elliptic Curve Cryptography* (ECC), which is based on the structure of an elliptic curve defined over a finite field and the difficulty of the so-called discrete logarithm problem for such elliptic curves. Compared to RSA, ECC uses smaller key sizes to get the same level of security.

2.4.1 Digital Signatures

A digital signature is used to guarantee the source and integrity of a message, i.e. that the message has been sent by the claimed sender and that it has not been altered [17]. RSA or ECC are commonly used for digital signatures.

A signature generation algorithm typically computes the hash value of the message m and then uses the hash value and the private key of the sender to obtain a digital signature. The signature is sent together with the message to the recipient.

To verify that the signature is correct, the recipient uses the public key of the sender and the received message and signature as inputs to a signature verification algorithm. If the message has been modified, i.e. the received message m' is not equal to the message m for which the signature was created, the signature verification will fail. A correctly verified signature proves that the sender is the owner of the private key corresponding to the public key used for the verification.

2.4.2 Public Key Infrastructure and Certificates

A *Public Key Infrastructure* (PKI) refers to the infrastructure that is needed to securely acquire public keys [17]. This typically involves the use of a *Certificate Authority* (CA) that is used to bind a public key to a certain entity. The CA, which is a trusted third party, verifies the identity of the entity and issues a digital certificate which contains the public key and some information about the key owner. The certificate is digitally signed by the CA.

2.5 JSON Web Token

JSON Web Token (JWT) is an open standard for secure information exchange between two parties [18]. The tokens use a compact, JSON-based format to include

all the required information. A JWT is digitally signed to make the information verifiable. Encryption is also possible.

A JSON Web Token is a string with three parts separated by dots. The first part is a header, the second part is the payload, and the third part is the signature, as shown in Figure 2.1. Each part is Base64Url-encoded.



Figure 2.1: The structure of a JSON Web Token

JSON Web Tokens are commonly used in an URL, as a HTTP POST parameter, or in the HTTP header to include information about a logged-in user [19]. It is also a good way of securely transmitting other information between different entities.

JWT is a JSON-based alternative to the XML-based *Security Assertion Markup Language* (SAML) 2.0. SAML has some extra features not supported by JWT, but on the other hand SAML is much larger in size and more complex. JWT should be used when compactness and simple implementation are important considerations, but it is not considered to be a full replacement for SAML assertions [18].

2.5.1 Header

The header format is defined in the *JSON Web Signature* (JWS) standard [20].

Two header parameters are typically included for a JWT:

- "alg" (Algorithm) – identifies the digital signature algorithm or *Message Authentication Code* (MAC) algorithm used to secure the JWT.
- "typ" (Type) – declares the type, which always is "jwt" for JSON Web Tokens.

An example header, declaring that ES256 (*Elliptic Curve Digital Signature Algorithm*, ECDSA, using the SHA-256 hash algorithm) is used for the digital signature of the JWT:

```
{
  "typ": "JWT",
  "alg": "ES256"
}
```

The header is Base64Url-encoded, which results in the following string:

```
eyJ0eXAiOiJKV1QiLCJhbGciOiJIJFUzI1NiJ9
```

2.5.2 Payload

The payload part of a JWT contains claims, which are statements about an entity (for example a user) and other metadata [18]. A claim has a name and a value. The JWT standard has a list of registered claim names, but it is also possible to use other claims if both parties agree on using them.

Some registered claim names that are useful:

- "iss" (Issuer) – identifies the entity that issued the JWT.
- "sub" (Subject) – identifies the subject of the JWT, i.e. the entity that the claims are statements about.
- "aud" (Audience) – identifies the recipients that the JWT is intended for.
- "iat" (Issued At) – the time³ at which the JWT was issued.
- "exp" (Expiration time) – the expiration time³, after which the JWT should not be accepted.

An example payload, issued by `tomnil.se`, stating that Tomas Nilsson is an administrator:

```
{
  "iss": "tomnil.se",
  "exp": 1462689900,
  "name": "Tomas Nilsson",
  "admin": true
}
```

The payload is Base64Url-encoded, which results in the following string:

```
eyJpc3MiOiJ0b21uaWwuc2UiLCJleHAiOjE0NjI2ODk5MDAsIm5hbWUiOiJUb21hcyB0aWxzczI5uIiwiaWF0Ij0iOnRydWV9
```

2.5.3 Signature

The concatenation of the encoded header and the encoded payload (separated by '.') is used as input to the signature algorithm [20]. For example, if the elliptic curve based signature algorithm⁴ ES256 is used, the result of the digital signature is the elliptic curve point (R, S) . The signature included in the JWT is the concatenation $R || S$.

A Base64Url-encoded signature of the examples used for the header and the payload:

```
dVwifqdTEkGJ_qI4Msumk-CrBCnbXwwKu5qRgwC1auZHOGXxcpi1VcYwrH307_MFYCfzS827eM_5R9WJ1H1U8w
```

³Unix timestamp, i.e. the number of seconds since 1970-01-01 00:00:00 UTC, is used.

⁴See Section 2.4 for more information about elliptic curves and digital signatures.

Putting it all together, this results in the following JWT:

```
eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJpc3MiOiJ0b21uaWwuc2UiLCJleH  
AiOjEONjI2ODk5MDAsIm5hbWUiOiJUbn21hcyB0aWxzcz29uIiwiaWF0Ij0iOnRydWV9.  
dVwifqdTEkGJ_qI4Msumk-CrBCnbXwwKu5qRgwC1auZHOGXxcpi1VcYwrH307_MFYCf  
zS827eM_5R9WJ1H1U8w
```

Calvin – Merging Cloud and IoT

Calvin is an application environment developed by Ericsson Research to simplify the development of distributed Internet of Things (IoT) applications [21]. The current implementation is written in Python and is an open source project available on Github¹. The current version of Calvin is fully functional, but there is much ongoing work to add more features and improve the existing functionality [3].

Since this thesis is about extending Calvin with authorization functionality, an introduction to what Calvin is and how it works is required to fully understand the following chapters.

3.1 Distributed Cloud for IoT

What makes Calvin really interesting is that it enables a distributed cloud for IoT, i.e. it makes it possible to execute different parts of the application on different devices to enable execution where it is most beneficial [21]. Some parts of the application may require to be executed on certain devices, whereas other parts may benefit from having for example low latency or much computing power. The most preferable way to deploy such an app would be to distribute it over multiple devices, which also would enable the possibility of parallel processing.

There are a large number of heterogeneous devices with different hardware and different communication methods used in the Internet of Things. Calvin can handle details about communication protocols, distribution, and hardware, hence hiding the complexity for the application developer. Instead, the developer can focus on writing an application where things talk to things, no matter on what devices they are located. Sometimes the entire application may be executed on the same device, but at a later time parts of it may have been moved to other devices. Using Calvin, the application still looks the same for the developer no matter if one or many devices are used for the execution.

3.2 Applications and Actors

A Calvin application has a lifecycle which can be divided into four distinct phases [22]:

¹The Calvin repository can be downloaded from <https://github.com/EricssonResearch/calvin-base>

- Describe
- Connect
- Deploy
- Manage

To *describe* the functionality of an application, actors are used as building blocks. An *actor* could be a computation, a service, a device, or something else that performs a certain task that can be reused in many different applications. The only way for an actor to communicate with other actors is through its inports and outports, which basically are first in-first out queues of tokens (data elements).

To write an actor, the developer describes actions, input/output relations and conditions for when to trigger an action. A typical scenario is that an event or data received on the input ports trigger an action, which processes the data in some way and produces new tokens that are sent to the output ports, as shown in Figure 3.1. For this to work, an actor may store information in its internal state and it may also have certain requirements on the device where it is executed. An actor is unconcerned about where the input data comes from and where the output data is sent.

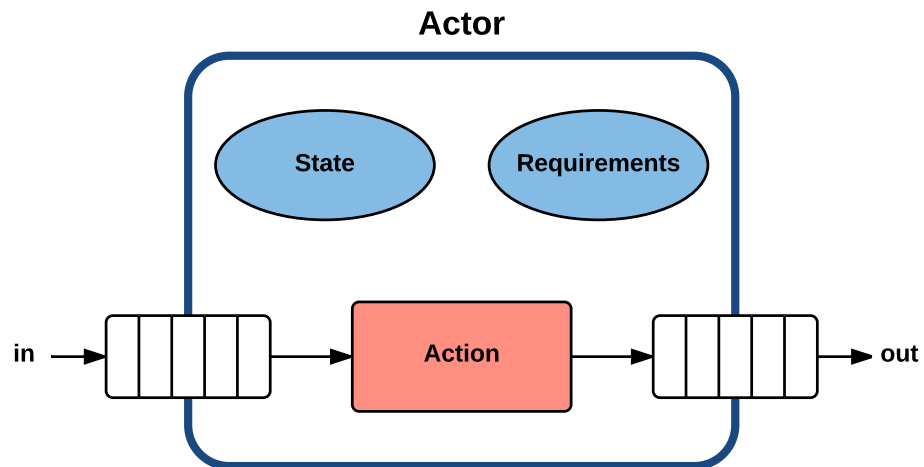


Figure 3.1: Actor model in Calvin

By *connecting* actors and hence forming a dataflow graph, an application is created. Complex applications can be constructed by combining many actors. A language called *CalvinScript* is used to describe which actors to use and how they are connected to each other. For a simple example of a Calvin script and its dataflow graph, see Section 3.6.

The *deploy* phase of the application lifecycle is when the application is instantiated according to the dataflow graph. In the current version of Calvin, applications can be deployed from the command line or by using a web interface. During this phase it is determined where each actor will start running.

The *managed* phase is entered by the application once it is running. Applications are monitored and different changes to the application may be handled automatically. As described more in detail in Section 3.4, the initial deployment decisions are not final. Instead, the actors can be moved to other devices later during their lifetime.

3.3 Runtime

A Calvin runtime is where actors execute. A broad range of physical devices, ranging from tiny sensor devices to large cloud servers, can be used for Calvin. One or several Calvin runtimes may be started on each physical device. The goal is to support as many platforms as possible, which means that alternative runtime implementations, adhering to the Calvin communication protocol and control commands, may be created in the future to support different microcontrollers [3].

Changing the code written by application developers is not required when moving execution to another runtime [21]. Instead, the complexity is hidden within the runtime implementation. A Calvin runtime controls execution of actors by handling for example data transport, message parsing, and scheduling, and provides an interface for access to runtime functionality from higher layers. The same runtime may (and usually will) concurrently handle several actor instances belonging to different applications, potentially deployed by different users.

To create a distributed execution environment, multiple runtimes need to be aware of each other and share information. For this, Calvin uses a global storage, implemented by using a *Distributed Hash Table* (DHT) or by letting a specific runtime handle storage for all runtimes [23]. The global storage contains information about the different runtimes and also an actor store with details about all the available actor implementations.

Different protocols can be used for the runtime-to-runtime communication depending on what is supported by the devices they are running on. The runtimes form a mesh network, in which actors can be migrated between runtimes [3]. How data is transported depends on the current locations of the actors in an application. From an application point of view, the network of runtimes creates an illusion of a single runtime. This means that writing distributed Calvin applications is as easy as writing non-distributed applications for an application developer.

3.4 Migration, Capabilities, and Requirements

Actors can be moved, or *migrated*, from one runtime to another after deployment of an application without interrupting the execution of the application. In most cases actor migration is almost instantaneous. A user can manually initiate migration by sending a command to the current runtime using the Calvin Control API (*Application Programming Interface*). However, a more interesting feature is the possibility of automatic actor migration, triggered by the runtime itself, to fulfill requirements or move the execution to a runtime where it is more beneficial [22].

Each runtime has a set of capabilities, i.e. functionality that can be used by actors executing on that runtime. An actor may require certain capabilities to

perform its intended action, for example an actor for file reading requires access to file handling functionality [24]. Runtimes try to automatically map the actor requirement list against capabilities provided by available runtimes and migrate actors accordingly.

An application is deployed to one of the runtimes, where it is instantiated and all actors are connected locally. Based on actor requirements and runtime capabilities, this might be followed by migration of some of the actors to other runtimes in the distributed execution environment [3]. Additionally, further requirements can be specified by the user when deploying an application, for example specifying that some actors must run on runtimes belonging to a certain organization or located in a certain country. Those requirements are matched against attributes describing the available runtimes, and are also taken into account when migrating actors after deployment of an application [25].

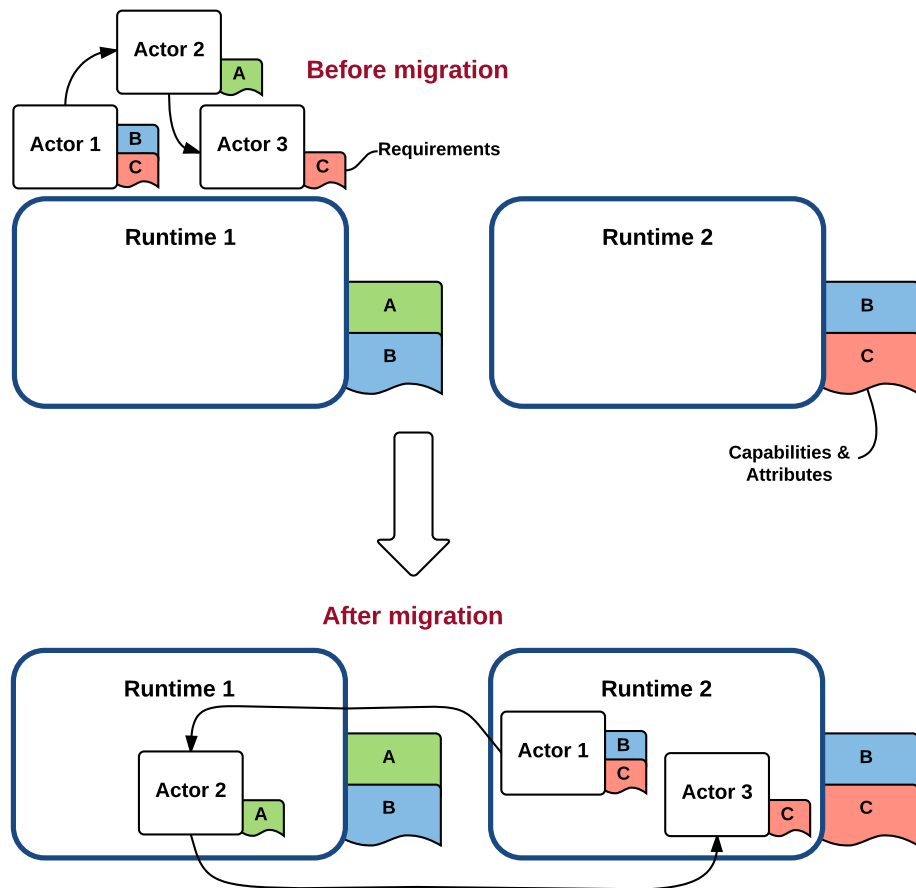


Figure 3.2: Calvin migration at deployment time based on requirements/capabilities matching

Figure 3.2 shows how actors are migrated based on their requirements when an application is deployed. The application, which consists of three actors, is deployed to Runtime 1. Actor 1 requires the capabilities B and C, and since Runtime 1 does not have capability C, Actor 1 is migrated to Runtime 2, which has the desired capabilities. For the same reason, Actor 3 also needs to be migrated, whereas Actor 2 remains on Runtime 1 since its requirements are fulfilled.

Migration of actors is possible since an actor can be considered in isolation. The application does not have a state that is shared by all its actors. The only data that has to be sent between the two runtimes during a migration is a serialized version of the internal state of the actor instance, what actor type it is, and information about its connections. The actor implementation itself is not sent between the runtimes, only what type of actor it is. A new actor of that type is instantiated on the target runtime and its ports are connected in the same way as before the migration. The implementation of the actor could be different on the target runtime, but the task it performs and the interpretation of the actor state must be the same. By deserializing the received internal actor state and using it as the state in the new actor, execution can continue where it left off before the migration [21].

3.5 Security

This thesis work improves the security in Calvin, but already since before Calvin had some security features that have been used as a part of the authorization solution presented in this thesis.

Verification of application signatures and actor signatures using OpenSSL is one feature that already was implemented. This includes the use of a truststore, where trusted certificates that are allowed to sign code are stored.

Another supported feature is the possibility to enter user credentials when deploying an application and authenticate the user using either a RADIUS server or a local file containing allowed username/password combinations. Simple runtime authorization for applications and actors based on user identities was also implemented, but the authorization part has now been replaced with the more advanced attribute-based authorization proposed in this thesis.

Parallel to the work on this thesis, there has been ongoing work on other security features in Calvin, for example securing the Distributed Hash Table using runtime certificates and signatures.

3.6 Example

An example is useful to get a better understanding of how Calvin works. The Calvin script and the dataflow graph for a rather simple Calvin application are shown in Figure 3.3.

The application, which is used to take a photo and show it on a screen, consists of three actors:

- A button (actor type: `io.GPIOReader`) to decide when to take a photo

- A camera (actor type: `media.Camera`) to take a photo
- A screen (actor type: `media.ImageRenderer`) to show the photo

```
button : io.GPIOReader(gpio_pin=23, edge="b", pull="d")
camera : media.Camera()
screen : media.ImageRenderer()

button.state > camera.trigger
camera.image > screen.image
```

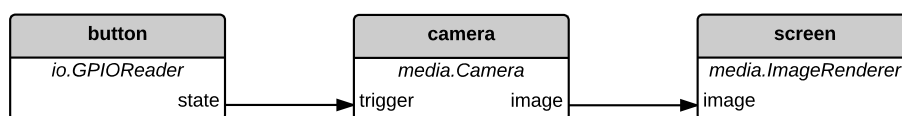


Figure 3.3: A Calvin script and dataflow graph example

The state output of the button actor is connected to the trigger input of the camera actor, which means that a button press will trigger the camera to take a photo. Similarly, the image output from the camera actor is sent to the input port of the screen actor, which displays the photo.

If the application is deployed to a runtime on a computer without a camera, the camera actor will not work. However, the application can still be deployed, but the camera actor will not be able to deliver the intended functionality and will instead be instantiated as a *shadow actor* that later can be migrated to suitable runtimes when they become available [22]. If the runtime is aware of another runtime on another device which has the required camera functionality, the camera actor will be migrated to that runtime, as explained in Section 3.4.

The capabilities needed by the actor that displays the photo might be present on many available runtimes. If the requirements are satisfied on the runtime where the application was deployed, the actor may stay there instead of being migrated. However, as mentioned in Section 3.4, it is possible to specify additional application deployment requirements, stating that for example the actor named `screen` should be placed on a runtime with a certain name or location [25]. These requirements will trigger a migration of the `screen` actor when the application is deployed.

Designing and Implementing Authorization in Calvin

The authorization design that is proposed in this chapter, and that also has been implemented in Calvin as part of this thesis work, enables very flexible attribute-based access control using a compact JSON-based language and also utilizing JSON Web Tokens to secure the communication.

The entities and the dataflow used in the XACML standard have been used as inspiration for the design of the authorization in Calvin, and hence much of the XACML terminology will be used to describe the design.

Figure 4.1 shows the authorization flow that has been implemented in Calvin. This chapter explains in detail how the different parts of the authorization flow have been designed and implemented.

4.1 Attribute-Based Access Control

To be able to grant access not only based on user identities but also based on other attributes such as information about actor requirements and the current environment, an attribute-based access control (ABAC) model was selected. By using policies that grant access depending on certain attributes, it is possible to finely tune the access decisions.

The policies can be very detailed, controlling access to certain resources on a specified runtime for a specific user, but it is also possible to write much more general policies, responsible for multiple runtimes and controlling access for many different users, applications and actors. Flexibility is a key word for the design of the authorization. The authorization is designed in such a way that any attributes can be used. If an attribute is specified in a policy and that attribute cannot be found for the authorization request, access will always be denied.

The attributes have been divided into four groups to structure them in the same way as in the XACML standard:

- Subject attributes
- Resource attributes
- Action attributes
- Environment attributes

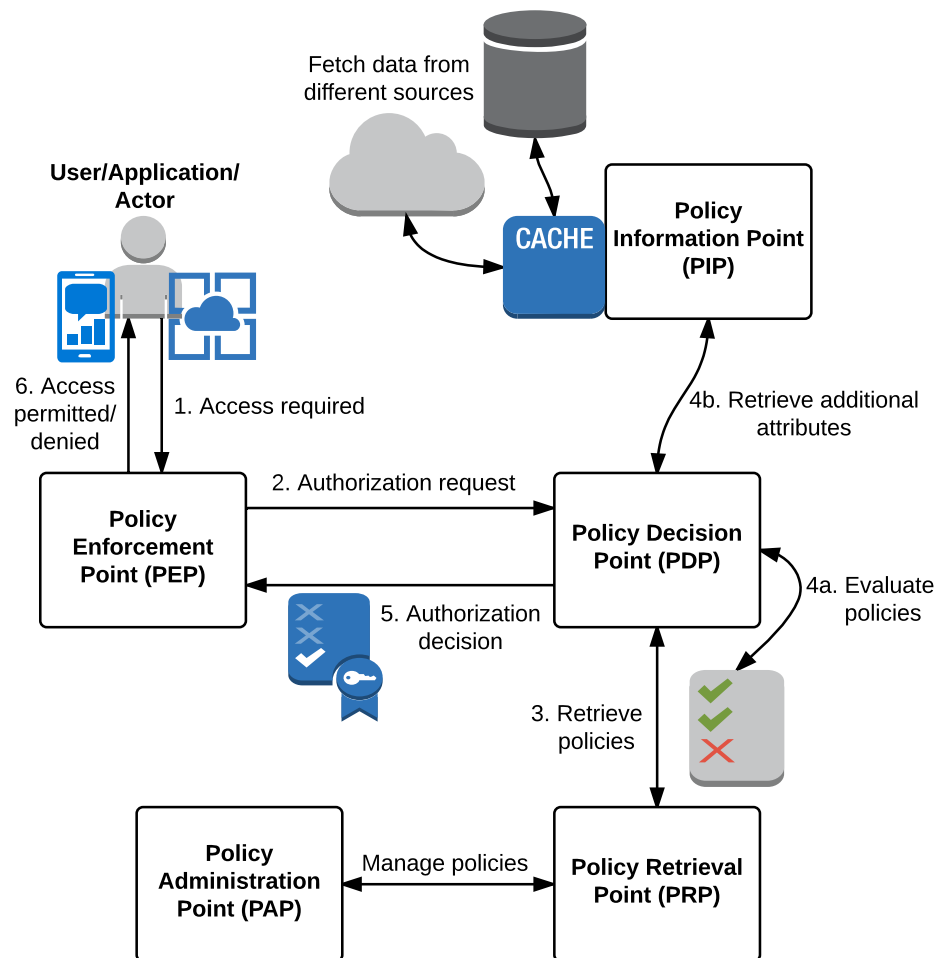


Figure 4.1: New authorization flow implemented in Calvin

4.1.1 Subject Attributes

In the computer security world, the term *subject* usually refers to the entity that requests access to a resource. The subject may be a person, service, or any other kind of entity [9]. A natural mapping of this to Calvin would be to define the subject as the user that deploys the application, as well as the application itself and the actors that belong to the application.

In Calvin, attributes describing the subject are retrieved from the authentication service. A user supplies credentials (username and password) when deploying an application. Depending on the runtime configuration, these credentials are either checked against a local file or against an external RADIUS server. If the authentication is successful, i.e. the username and password are correct, information about the user is returned and is used as subject attributes for future authorization requests.

As mentioned in Section 3.5, applications and actors may be signed by a trusted party. The signature is verified before an application or actor is started, and information about the signer may also be used as subject attribute.

Subject attribute examples for Calvin:

- User information
 - First name
 - Last name
 - Age
 - Nationality
 - Company
 - Department
 - Roles/groups
 - Email address
- Actor/application signer

4.1.2 Resource Attributes

The *resource* is the object that the subject wants to access in some way. In this context, the resource is a Calvin runtime. When a Calvin runtime is started it is possible to supply attributes describing the runtime.

Resource attribute examples for Calvin:

- Owner information
 - Organization
 - Role
 - Name of responsible person/group
- Address information
 - Country
 - City
 - Street
 - Building
 - Floor
 - Room
- Runtime information
 - Organization
 - Name of runtime/node
 - ID of runtime/node
 - Purpose (e.g. test, production)

4.1.3 Action Attributes

The *action* is what the subject wants to perform or access on the resource. Calvin actors always need basic runtime execution access. An actor may also have a list of functionality that is required on the runtime where it executes, for example file handling or camera functionality.

Action attribute examples for Calvin:

- Actor requirements
 - Basic runtime execution access
 - File handling (read/write)
 - Media functionality (e.g. camera, media player, image viewer)
 - Timer
 - Network functionality (client, server)
 - Sensors (e.g. distance, temperature, pressure)

4.1.4 Environment Attributes

Environment attributes here refer to attributes dealing with the current state or situation in which a request should be handled.

Environment attribute examples for Calvin:

- Current time
- Current date

Location from where access is requested and the communication channel type (protocol, encryption strength, etc.) are other typical environment attributes that may be used when such functionality is added in Calvin.

4.2 Policy Enforcement Point (PEP)

The Calvin runtime to which an application is deployed acts as *Policy Enforcement Point* (PEP). As part of this thesis work, the deployment process of an application has been changed to improve the way security is handled in Calvin. The new deployment process when security is enabled is shown in Figure 4.2.

When the user has been authenticated and the application signature has been verified, the PEP sends an authorization request to a *Policy Decision Point* (PDP) to determine if the application should be allowed to run. Similarly, an authorization request is sent for each actor in the application, when the actor signature has been verified, to check if the actor should have permission to execute and use the required runtime capabilities.

The procedure is similar when an actor is migrated. When the migration data is received by the target runtime, i.e. the runtime to which the actor is migrated, the implementation of the actor type is looked up and the actor signature is verified. The target runtime is the one that now acts as PEP and sends an authorization request to its PDP to check if access should be granted to the actor.

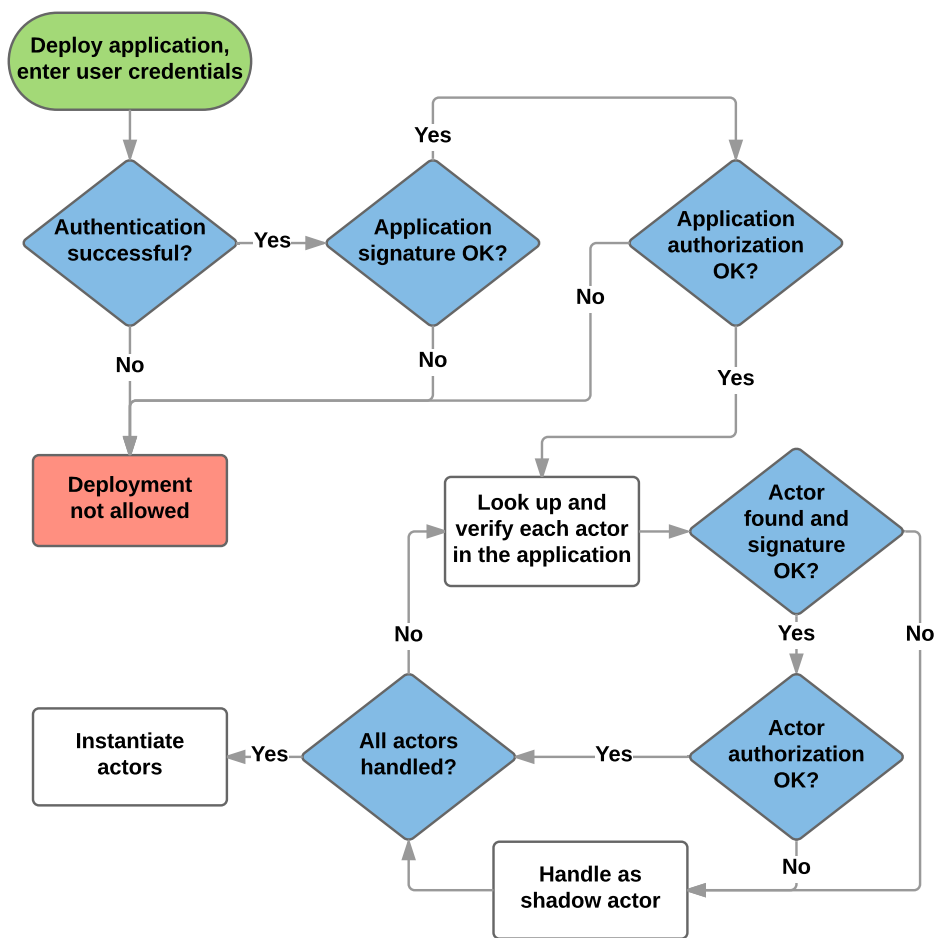


Figure 4.2: New deployment process in Calvin

New configuration options have been added to Calvin to allow a runtime to have either a local or an external PDP. A local PDP means that the PEP and the PDP are located on the same runtime, whereas an external PDP means that a PDP on another runtime acts as authorization server and makes authorization decisions on behalf of the PEP runtime.

4.2.1 Runtime Registration

When the PEP runtime is started, a JSON representation of the runtime attributes (see Section 4.1.2) is sent to the PDP to enable future authorization. An example of what is sent in this runtime registration is shown in Appendix A. The PDP saves the runtime attributes together with the runtime ID¹. As a result of the registration process, it is enough to send the runtime ID in future authorization requests instead of all the resource attributes.

4.2.2 Authorization Requests/Responses

JSON is used for authorization requests and responses sent between the PEP and the PDP. A request contains subject attributes returned from the authentication service (see Section 4.1.1), and the runtime ID of the PEP. For actor authorization requests, a list of required runtime functionality is also included in the request. An example request is shown in Appendix B.1. The format reminds of the request/response format used in the XACML JSON Profile (see Section 2.3.5), but a more compact JSON representation is used here.

The response from the PDP includes an authorization decision (`permit`, `deny`, `indeterminate` or `not_applicable`). A response with the decision `permit` indicates that access should be granted by the PEP, whereas all other decisions should result in denied access. Figure 4.2 shows how the authorization decision affects the steps taken by the PEP runtime when an application is deployed.

If the decision is `permit`, the response may also include constraints under which the authorization decision is valid. These constraints are included in a section of the response called `obligations`². An example response can be found in Appendix B.2. The PEP uses local authorization check plugins to continuously check the constraints when an actor wants to perform an action after instantiation. New authorization check plugins can easily be added to Calvin. The plugin `time_range` has been implemented to make it possible for an authorization decision to be valid only between a specified start and end time each day.

4.2.3 JWT for External Authorization

If the PDP is external, signed JSON Web Tokens (JWT) are used for runtime registration, authorization requests, and authorization responses to secure the in-

¹Calvin uses randomly generated UUIDs (universally unique identifier) as runtime IDs, e.g. `a77c0687-dce8-496f-8d81-571333be6116`. If runtime certificates are used, the ID is specified in the certificate, which is signed by a trusted Certificate Authority (CA).

²The term *obligation* is used in the XACML standard for an operation that should be performed by the PEP when enforcing an authorization decision [12].

formation exchange between the two runtimes where the PEP and the PDP are located³. The JWT is used in almost the same way as SAML is used in the XACML SAML Profile (see Section 2.3.4).

The following claims (most of them are registered JWT claim names, see Section 2.5.2) are used in the JWT payload:

- **"iss"** (Issuer) – the ID of the runtime that creates the JWT.
- **"sub"** (Subject) – the ID of the actor⁴ that the request/response applies to (only used for actor authorization requests/responses).
- **"aud"** (Audience) – the ID of the runtime to which the JWT is intended.
- **"iat"** (Issued At) – the time at which the JWT was issued.
- **"exp"** (Expiration time) – the expiration time for the JWT (may be configurable in the runtime settings in future Calvin releases).
- **"attributes"/"request"/"response"** – the runtime attributes (see Appendix A) or the authorization request/response (see Appendix B), i.e. the information that would have been sent if a local PDP would have been used.

The runtime that creates the JWT uses an elliptic curve private key to create the digital signature. When the JWT is received by the other runtime, the signature is verified using the corresponding public key of the sending runtime. If the signature is invalid, the information in the JWT will not be accepted.

Similarly, the JWT will also be rejected if the runtime ID specified in the **"aud"** claim is not the same as the ID of the runtime that has received the JWT, or if the expiration time has passed. When an authorization response is received, it is important that the runtime ID in the **"iss"** claim corresponds to the runtime ID of the specified authorization server, and that the **"sub"** claim contains the ID of the correct actor when such an ID has been included in the request.

The public key of the sender is available to the recipient as part of a runtime certificate. A public key infrastructure, where runtime certificates signed by a trusted CA are issued and distributed to other Calvin runtimes, has been created as part of other security-related work in Calvin and is therefore not explained in detail in this thesis. The important part here is that the runtime that creates the JWT must have a private key to sign the JWT, and that the receiving runtime that wants to verify the JWT signature must have the corresponding public key, as part of a runtime certificate signed by a trusted CA.

4.3 Policy Decision Point (PDP)

The *Policy Decision Point* (PDP) is a Calvin runtime that evaluates requests from a PEP against authorization policies and returns an authorization decision. An example policy can be found in Appendix C.

³The following JSON Web Token implementation in Python has been used:
<https://github.com/jpadilla/pyjwt/>.

⁴In the same way as for runtime IDs, Calvin uses randomly generated UUIDs (universally unique identifier) as actor IDs, e.g. `fc36ce29-e72d-4b1c-9447-4cb9813708f6`.

4.3.1 Find Matching Policies

Policies are retrieved from a *Policy Retrieval Point* (PRP). The format of a policy reminds of XACML policies, but JSON is used instead of XML.

Each policy has a `target` section, which is used to decide if the policy is applicable to the incoming authorization request. If all the attribute values in the `target` section of the policy match the attribute values in the authorization request, the policy is further evaluated, whereas the policy is ignored if any of the attribute values do not match. A policy without a `target` section is applicable to any request.

To decrease the number of policies needed, it is possible to use a list of many alternative values for a certain attribute in the policy. Regular expressions⁵ can also be used to define a pattern that has to be found in the attribute value in the request instead of defining the entire string. Some simple examples are given here to explain the usefulness of these features.

The attribute `"first_name"` matches if the attribute value in the request is `"Tomas"` or `"Gustav"`:

```
{"first_name": ["Tomas", "Gustav"]}
```

The attribute `"email"` matches if the attribute value in the request ends with `"@ericsson.com"`:

```
{"email": ".*@ericsson.com"}
```

The attribute `"requires"` matches if the attribute value is exactly `"runtime"` or exactly `"calvinsys.events.timer"` or begins with `"calvinsys.io"`:

```
{"requires": ["runtime", "calvinsys.events.timer", "calvinsys.io.*"]}
```

4.3.2 Evaluate Policies

If a policy `target` matches the request, the complete policy will be evaluated. The policy has a `rules` section with one or many rules that are evaluated to get a policy decision. Each policy rule returns one out of four possible decisions for an incoming request: `permit`, `deny`, `indeterminate` or `not_applicable`. A rule combining algorithm (currently `permit_overrides` or `deny_overrides`) is specified in the policy to determine how the rule decisions are combined into a policy decision.

A policy rule can contain a `target` section, in the same way as the `target` for the entire policy, to specify for which requests that rule is applicable. In addition to a `target`, a rule can also have a `condition` section, which can be used for a more advanced form of attribute comparison using different functions.

The functions in Table 4.1 can be used as `function` in the `condition` section of a policy, and more functions can easily be implemented in the future. Each function requires a number of `attributes` as function arguments. The output of a function is `true` or `false`.

⁵Since Calvin is implemented using Python, the regular expression syntax described in the Python documentation (<https://docs.python.org/2/library/re.html>) is used.

Table 4.1: Functions that can be used in a condition in a policy

Function name	Number of arguments
<code>equal</code>	2
<code>less_than_or_equal</code>	2
<code>greater_than_or_equal</code>	2
<code>not_equal</code>	2
<code>and</code>	≥ 2
<code>or</code>	≥ 2

Input arguments of a function can be either constant values (regular expressions can be used), a list of such constant values, or a reference to an attribute in the authorization request. A string that starts with `"attr:"` is a reference to an attribute value in the request, e.g. `"attr:resource:address.country"` (where `resource` is the argument type and `address.country` is the argument name). Missing attributes, i.e. referenced attributes that are not found in the authorization request, are fetched from a *Policy Information Point* (PIP) if possible (e.g. the current date in the example below).

The following example of a `condition` in a policy shows how functions can be nested in two levels, in this case an `"and"` function using the result of an `"equal"` function and a `"greater_than_or_equal"` function as input arguments:

```
{
  "condition": {
    "function": "and",
    "attributes": [
      {
        "function": "equal",
        "attributes": ["attr:resource:address.country",
                     ["SE", "DK"]]
      },
      {
        "function": "greater_than_or_equal",
        "attributes": ["attr:environment:current_date",
                     "2016-03-04"]
      }
    ]
  }
}
```

In the example `condition` above, the `"equal"` function looks at the runtime attributes to determine if the runtime is located in Sweden (SE) or Denmark (DK), and the `"greater_than_or_equal"` function checks if the current date is 2016-03-04 or later. If both of these functions return `true`, the result of the `"and"` function will be `true`.

If the rule is satisfied, i.e. the result when evaluating the `condition` is `true`,

the specified rule **effect** (**permit** or **deny**) will be returned as rule decision and potentially be combined with decisions from other rules in the policy to get a policy decision. Otherwise, if the rule is not satisfied, **not_applicable** will be returned as the rule decision. If any errors occur while evaluating a policy, the decision will be **indeterminate**.

A rule may contain an **obligations** section with constraints under which the authorization decision is valid. Obligations can only be used if the rule decision is **permit**. If other policies also return **permit** but without obligations, the obligations will not be taken into consideration since a **permit** decision without constraints has been found. Otherwise, the obligations will be included in the response and will be handled by the PEP (see Section 4.2.2).

The following example of an **obligations** section in a policy rule specifies that the **permit** decision is only valid between 09:00 and 17:00 every day:

```
{
  "obligations": [
    {
      "id": "time_range",
      "attributes": {
        "start_time": "09:00",
        "end_time": "17:00"
      }
    }
  ]
}
```

4.3.3 Combine Policy Decisions

If multiple policies match the request, the policy decisions are combined into a combined policy decision using one of the combining algorithms **permit_overrides** or **deny_overrides** (which one is specified in the configuration for the PDP). The combined policy decision is returned as authorization decision to the PEP.

If **permit_overrides** is used as combining algorithm, the combined policy decision is determined in the following way:

1. If any of the policy decisions is **permit**, the result will be **permit**.
2. Otherwise, if any of the decisions is **indeterminate**, the result will be **indeterminate**.
3. Otherwise, if any of the decisions is **deny**, the result will be **deny**.
4. Otherwise, the result will be **not_applicable** (this will also be the result if the request does not match any policies).

The procedure for the **deny_overrides** algorithm is obtained by letting the words **permit** and **deny** change places in the above list.

A useful example in many situations is to use `permit_overrides` as combining algorithm and always have a fallback policy which denies all requests. If no matching policies that return `permit` have been found, the fallback policy will make sure that the decision will be `deny`.

4.4 Policy Information Point (PIP)

If an attribute used in a policy `target` or `condition` is not included in the authorization request, the PDP will ask the *Policy Information Point* (PIP) for the value of the attribute.

Environment attributes, such as the current date or current time, are usually not included in the authorization request, but can easily be obtained from the PIP. For some attributes the PIP may use another attribute that was provided in the authorization request from the PEP to obtain the requested attribute. An example of that is if the authorization request includes a reference to an actor in the actor store, but the policy needs the name of the actor signer to make a decision. Then the PIP can use the actor store reference to get information about the actor signer from the actor store.

The PDP and the PIP are located on the same Calvin runtime, but the PIP may use external requests to obtain the requested attribute value. An attribute cache has been implemented in the PIP to prevent the same attribute value from being computed or fetched more than once per authorization request. If an attribute value received from the PIP is referenced more than once when a request is evaluated against policies, the cache makes it faster to get the attribute value and also ensures that the same value will be used for all evaluations when handling the authorization request.

4.5 Policy Retrieval Point (PRP)

A *Policy Retrieval Point* (PRP) is where the authorization policies are stored. A PRP must implement methods for creating, updating, deleting, and retrieving policies. An abstract Policy Retrieval Point class has been written to define the methods that need to be implemented by a PRP. In the current implementation, the file system is used as PRP. Policies are stored in JSON files on the same machine as the PDP runtime. The directory path and a name pattern for the JSON files are specified in the runtime configuration.

4.6 Policy Administration Point (PAP)

A *Policy Administration Point* (PAP) is used to create, modify or delete policies. The Calvin Control API has been extended with a REST API for policy management interactions between the PAP and the PRP, see Table 4.2.

To make policy creation and editing easy for the administrator, it is a good idea to create a user-friendly policy tool instead of writing JSON directly. Parallel to my work on the authorization in Calvin, my advisor at Ericsson has created a

web-based policy tool, which can act as a PAP. In the web tool, the user chooses between available functions and attributes and enters desired attribute values in different text fields. The policy tool translates the form input data to the correct JSON format and uses the REST API to create or update the policies.

Table 4.2: REST API for policy management

URI	HTTP method	Action
/authorization/policies	POST	Create new policy (input: JSON policy)
/authorization/policies	GET	Get all policies
/authorization/policies/<id>	GET	Get policy <id>
/authorization/policies/<id>	PUT	Update policy <id> (input: JSON policy)
/authorization/policies/<id>	DELETE	Delete policy <id>

4.7 Smart Migration

If an actor authorization response contains an **obligations** section with constraints under which the authorization decision is valid, those constraints will be checked locally by the PEP, as explained in Section 4.2.2.

The constraints may for example be that the actor only is granted access between 09:00 and 17:00 every day. If the actor is started at 15:00, it will run for two hours before access is denied. At 17:00, when access is denied, the runtime will automatically try to migrate the actor to another runtime. To prevent unsuccessful migration attempts, the current runtime will try to make sure that the actor has permission to run on the runtime to which it tries to migrate the actor. This process is here called *smart migration*.

The following steps, also illustrated in Figure 4.3, are involved in a smart migration of an actor instance in Calvin:

1. When access is denied, the Camera actor will stop running on Runtime 1.
2. Runtime 1 asks global storage for a list of possible migration destinations that satisfy the requirements.
 - The requirements consist of both user-specified requirements (e.g. that the user accepts that the actor runs on Runtime 1-4), and actor requirements (capabilities needed by the runtime to execute the actor).
 - A list of runtimes that satisfy the requirements can be retrieved from global storage, which also has information about each runtime's authorization server, i.e. where the PDP is located.
3. Runtime 1 sends an authorization search request to Runtime 2, which is the PDP responsible for the first runtime on the list of possible migration destinations.

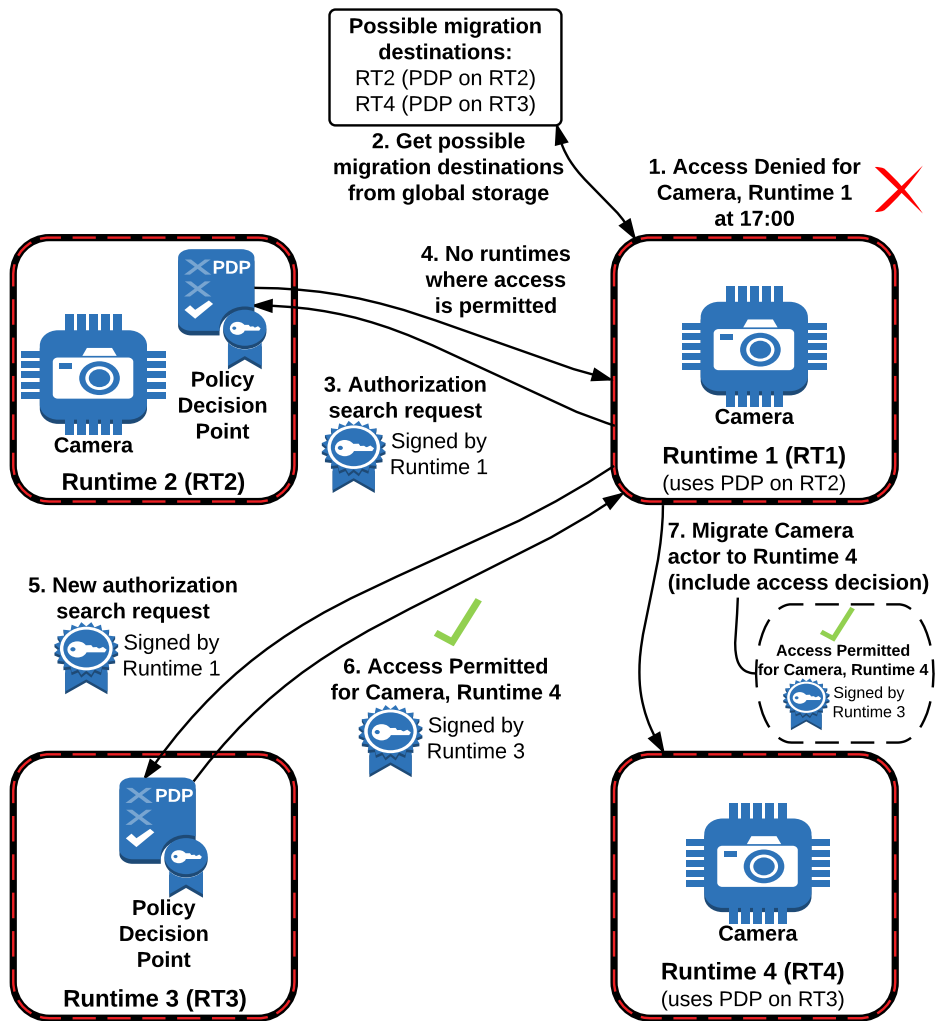


Figure 4.3: Smart migration when access is denied.

- The authorization search request is a signed JWT which reminds of a normal authorization request (see Section 4.2.2) and which also includes the list of possible migration destinations. Subject attributes are included in the request, but no resource attributes are included since Runtime 1 does not know attributes of other runtimes. A reference to the actor in the actor store is the only information about the actor in the request.
4. The PDP on Runtime 2 will add runtime attributes to the request and evaluate the request against policies in the same way as when a normal authorization request arrives. In this example, the response to Runtime 1 is that no runtime where access is permitted was found.
 - All runtimes that use the PDP on Runtime 2 have registered their attributes there. Hence, the PDP can add that information to the authorization search request.
 - The PDP will evaluate the request against policies for all runtimes that have been registered there and which also are included on the list of possible migration destinations in the authorization search request.
 - Actor information, such as actor signer, is not included in the authorization search request since the actor implementation and signature may be different on different runtimes. Instead, the PIP will look up information about the actor using the actor store reference.
 5. If no runtime was found by the first authorization server, Runtime 1 will send a new authorization search request to the next authorization server on the list, which in this example is Runtime 3.
 6. Runtime 3 finds one runtime (Runtime 4) where access is permitted and creates a signed JWT with an authorization response that looks exactly like a normal authorization response (see Section 4.2.2).
 - The "aud" claim in the JWT is set to the ID of Runtime 4 to indicate that the decision is intended for Runtime 4.
 7. Runtime 1 will initiate a migration of the Camera actor to Runtime 4 and include the signed access decision JWT from Runtime 3 in the migration info sent to Runtime 4.
 - When Runtime 4 receives the migration info, it can check that the authorization decision is signed by Runtime 3 (its authorization server) and that the decision is intended for Runtime 4 and has not expired. Hence, there is no need for Runtime 4 to send a new authorization request to its PDP. Instead, the actor can start executing directly.

Authorization Tests in Calvin

This chapter presents test results for the authorization that has been implemented in Calvin. Both the correctness and the performance of the implementation have been tested. For details about the implementation and explanations of the terms used to describe the tests, see Chapter 4.

5.1 Correctness Tests

A Python testing tool called *pytest*¹ has been used to test the correctness of the authorization implementation. The tests presented in Table 5.1 do not cover all possible use cases since the implemented authorization is very flexible and can be used in many different ways. However, since the tests deal with all the functionality in some way, there is a high probability that the authorization works in the intended way if all the tests pass. The tests have been performed with several runtimes running on the same machine.

A test is considered to be successful if the effect of the authorization corresponds to the expected application/actor behavior for the current authorization policies. This is tested by using different combinations of applications, user credentials and runtimes in the deploy command, and checking for example if an actor produces the expected output or if the entire application is prevented from being deployed. All tests do not have to be performed for both external and local authorization since they both use the same PDP implementation. The only difference is that JWTs are used for external authorization.

Table 5.1: Correctness tests and results for Calvin authorization

Test Description	Expected Result	Test Result
Attribute in request not accepted by policies (application authorization)	Deployment denied	✓
Attribute in request not accepted by policies (actor authorization)	No actor output	✓

Continued on next page

¹For more information about *pytest*, see <http://www.pytest.org>.

Continued from previous page

Test Description	Expected Result	Test Result
Accepted attributes for both application and actors	Actor produces output	✓
Reference to attribute not present in request but retrievable from PIP	Actor produces output	✓
Reference to attribute not present in request and not retrievable from PIP (actor authorization)	No actor output	✓
Attribute included in list of accepted attributes in policy target	Actor produces output	✓
Attribute matches regular expression in policy target	Actor produces output	✓
No policies match the request (actor authorization)	No actor output	✓
Two policies without target (matching all requests), one permitting, the other denying (using <code>permit_overrides</code>)	Actor produces output	✓
Two policies without target (matching all requests), one permitting, the other denying (using <code>deny_overrides</code>)	Deployment denied	✓
Policy with "and" function using results from functions "equal", "not_equal", "less_than_or_equal", "greater_than_or_equal" that should all evaluate to true	Actor produces output	✓
Policy with "or" function using results from functions "equal", "not_equal", "less_than_or_equal", "greater_than_or_equal" where none should evaluate to true	No actor output	✓
External authorization with correct JWT	Actor produces output	✓
External application authorization with invalid JWT signature in response	Deployment denied	✓
External application authorization with invalid "aud" claim in JWT response	Deployment denied	✓
External application authorization with expired "exp" claim in JWT response	Deployment denied	✓
Smart migration when access is denied due to obligations with time range	Actor is migrated to other runtime where it starts producing output	✓

5.2 Performance Tests

To analyze the performance of the authorization implementation, the execution time to get an authorization decision from a local PDP has been measured for different scenarios using Python's `timeit` module. One way to measure the scalability is to increase the number of policies that authorization requests are evaluated against. Another important factor for the performance is the usage of suitable `target` sections in the policies. If the `target` does not match, the policy will not be further evaluated, hence saving time.

The following tests have been performed to measure the performance:

- **Test 1:** The attributes of the authorization request do not match the `target` of any policy. Only the `target` of each policy will be checked, and not the complete policy.
- **Test 2:** The attributes of the authorization request match the attributes in each policy `target`, but the `condition` (consisting of three function evaluations) in each policy is not satisfied. All policies will be completely evaluated since the `target` matches.

Table 5.2 shows the test results for different number of policies. The policies have been written such that all available policies must be evaluated before returning a decision.

Table 5.2: Performance test results for Calvin authorization

Number of policies	Test 1: Execution time [s]	Test 2: Execution time [s]	Time Ratio Test 2/Test 1
10	0.0013	0.0016	1.23
50	0.0026	0.0037	1.42
100	0.0041	0.0060	1.46
500	0.0177	0.0274	1.55
1000	0.0343	0.0540	1.57
5000	0.1683	0.2684	1.59
10000	0.3804	0.5905	1.55

Using an external PDP instead of a local PDP requires less computing power and storage on the device running the PEP runtime. However, it adds a delay to the time it takes to get an authorization decision compared to what is presented in Table 5.2. Encoding and decoding JWTs take negligible time. Instead, the delay largely depends on the quality of the connection between the PEP and the PDP, i.e. the time it takes to send the JWTs between the two runtimes.

Test results, design choices, and security considerations for the authorization solution that has been implemented in Calvin are discussed in this chapter. Future work to further improve the authorization and related functionality in Calvin is also suggested.

6.1 Comments on Test Results

The test results presented in Chapter 5 show that the authorization implementation works as intended and performs well. Access is permitted correctly according to the policies, and the authorization decision results in the expected behavior for the actor or application. The implemented authorization satisfies the aim of enabling fine-grained decisions to decide exactly to whom access is granted, and the runtime is also able to perform post-instantiation access control and successfully migrate an actor to another runtime if access is denied. The implementation is fault tolerant by always denying access and continuing operation of the runtime properly when any errors occur, such as incorrect syntax for requests/responses or invalid signatures.

The performance test results show that the execution time to get an authorization decision is affected by the way policies are written and the configuration of the Policy Decision Point (PDP). Optimal performance is achieved by using suitable `target` attributes that result in few matching policies for a request, hence only requiring a few policies to be fully evaluated. This is confirmed in Table 5.2, which shows that the execution time is approximately 50% longer for the test if all policies have to be fully evaluated compared to if only the `target` in each policy has to be checked.

It is also important that the PDP is configured in a clever way so it uses a policy combining algorithm that works well for the policies that have been written. An example of a poorly configured PDP is to have `deny_overrides` as combining algorithm if only one policy matches the request and that policy evaluates to `permit` since `deny_overrides` means that all policies have to be checked in order to assure that no other policy evaluates to `deny`. Instead, if `permit_overrides` is used, the PDP will stop checking further policies and return the decision directly when the policy evaluating to `permit` has been found, which potentially has a great influence on the number of policies that have to be checked.

6.2 Design Choices

The decision to use an attribute-based access control (ABAC) model for Calvin was easy to make since it offers much more flexibility than other alternatives such as role-based access control (RBAC). Being able to base the authorization decision not only on who the user is, but also on attributes answering questions about what, when, where, why, and how the access is requested, makes it possible to control access in detail, which was one of the aims of the thesis. Gartner analysts have predicted that 70% of the enterprises will use ABAC as the dominant mechanism to protect critical assets by 2020 [26], which indicates that ABAC is the future.

6.2.1 JSON-based XACML

XACML is a dominant standard in the ABAC area, and the SAML profile of XACML describes a secure way to send authorization requests and responses. However, both these standards are XML-based, which often is considered to be too verbose for constrained Internet of Things devices. A better choice would be to use JSON since it is a much more lightweight and compact format, resulting in higher parsing efficiency and less data being sent. The trend to simplify standards and adapt them to IoT and cloud technologies has also reached the XACML committee, which by releasing the REST and JSON profiles has taken a step in that direction [27]. However, further simplification is desirable, such as JSON-based policies and a JSON-based replacement for the XACML SAML Profile.

The idea of the proposed solution in this thesis was to design a standards-based authorization that has been adapted to IoT and Calvin's distributed execution environment. As basis for the design, the XACML architecture has been used, but XML has been replaced entirely by JSON. Using JSON for both the requests and the policies also simplifies matching between requests and policies.

The JSON Profile of XACML only deals with JSON versions of XACML requests and responses, but there has also been attempts to convert the XML-based XACML policies to JSON [28]. However, these JSON versions of the requests, responses, and policies all look much like XML in JSON format, containing unnecessary elements to remind of the structure of their XML equivalents. The language for policies, requests, and responses proposed in this thesis uses a simplified and more compact JSON notation, which still easily can be converted to an equivalent XML-based version that is valid according to the XACML standard.

Since JWT in many aspects is the JSON version of what SAML does in the XML world [29], the proposed solution uses JWT to secure the authorization requests and responses. One of the main reasons why JWT signatures based on elliptic curve cryptography were chosen is that a public key infrastructure with elliptic curve public/private keys and runtime certificates already had been created for other security features in Calvin and could be used for the JWT as well.

6.2.2 Adaptable to Constrained Devices

To make the authorization solution adaptable to different situations and devices was an important design goal. If the aim is to minimize network traffic, the runtime can use a local PDP. The PAP can still be centralized and use the REST

API to push policy changes to many different local PDPs. If the device instead is constrained in storage or processing power, it might be better to use an external PDP that is less constrained.

6.3 Security Considerations

When a local PDP is used, there is no external network traffic involved in the authorization, so as long as the policies are correct there are no security threats to consider. However, if an external PDP is used as authorization server or an authorization decision is received from another runtime as part of a smart migration, there are several threats that must be considered and for which countermeasures have been implemented.

The use of object security in the shape of signed JSON Web Tokens (JWT) protects the authorization process from most of the threats. Since public/private key pairs are used for the signature, the recipient can be sure that the sender is the one it claims to be. The JWT signature is also used to verify the integrity of the data, i.e. to make sure that the JWT payload has not been altered.

To prevent an attacker from redirecting an authorization response to another runtime which uses the same authorization server, the JWT contains the ID of the intended recipient in the "aud" claim. The actor ID in the "sub" claim is used to indicate for which actor instance the response is valid.

Replay attacks are prevented by using the "exp" claim to include an expiration time for the JWT. By having a short valid lifetime, the JWT from an authorization server cannot be sent again to get access at a later time. The expiration time is also useful for smart migrations to make sure that the authorization decision that is included in the migration info has been issued recently. It is assumed that the different runtimes have synchronized clocks. A possible alternative or complement to the expiration time is to use the "jti" (JWT ID) claim to give each JWT a unique identifier. If the recipient stores recently received JWT IDs and compare these IDs with the ID of the incoming JWT, replay attacks will be prevented.

Denial of service, in this case making the authorization server unavailable by sending a large number of requests at the same time, is a possible threat. Specifying multiple alternative authorization servers for a PDP could be a suitable countermeasure if the probability of such an attack is high.

Subject attributes are retrieved during the authentication process by the runtime where the application is deployed. If an actor is migrated to another runtime within the same domain, it is assumed that the runtime trusts the first runtime when it comes to authentication of the subject attributes. There is currently ongoing work on how to handle subject attributes when crossing domain boundaries.

6.4 Privacy Considerations

Attributes sent to an external PDP may contain privacy-sensitive information. If that is the case, encryption of the data would be desirable to prevent eavesdropping. *Transport Layer Security* (TLS) will soon be added to Calvin, which adds

encryption on the transport layer. An alternative way to prevent the information from being disclosed is to encrypt the signed JWT.

The user can specify in the application requirements to which runtimes migration is permitted. Hence, the subject attributes will never be sent to any runtimes not accepted by the user.

A response from an authorization server does not include the reason why access was granted or denied, which means that potentially secret details about what the policies look like are not revealed.

6.5 Future Work

A fully functional authorization solution has been implemented in Calvin, but there are a number of areas where future work is desirable.

The following tasks have been identified for future work to further improve the authorization that has been presented in this thesis:

- Implement REST API authentication to protect policy management (this is not only a desired feature for the authorization part since the API also is used for controlling other parts of Calvin).
- Write alternative Policy Retrieval Point (PRP) implementations, for example a database PRP where policies are indexed based on the attributes in their `target` section to further speed up the authorization process.
- Implement more features from the XACML standard, for example other functions and the possibility to use more than two levels of nested functions.
- Extend smart migration to work for authorization servers in other domains. The main problem with domain boundary crossing is how to handle translation of subject attributes into meaningful attributes in the target domain.
- Handle revocation of access and other changes to policies. A possible approach could be to inform all runtimes when the authorization server policies have been updated.
- Add possibility to encrypt the authorization requests/responses and/or use TLS.
- Implement more plugins to allow other constraints under which an authorization decision is valid, e.g. constraints based on the current runtime location (if it is movable) or the current temperature.
- Port the Python implementation of the authorization to other languages when for example versions of Calvin for more constrained devices are written.

This thesis proposes an authorization framework that enables fine-grained attribute-based access control using compact message and policy formats based on JSON. The combination of using a lightweight message and policy format while still being able to make access decisions based on many different attributes is one of the main differences between this solution and other existing solutions. This makes it highly suitable for the dynamic distributed execution model in Calvin.

The proposed authorization solution has successfully been implemented in Calvin and is now a part of the Calvin repository available on Ericsson Research's Github¹.

Performance and correctness tests show that the authorization works well. The flexibility of the solution makes it adaptable to different environments, for example by allowing more powerful devices to evaluate authorization requests on behalf of constrained devices. It has also been shown that the authorization framework can be used to enable smart migration decisions in Calvin.

The authorization implementation meets all of the aims that were set before starting the research. Future work on the authorization to further enhance its performance is encouraged. The authorization framework has been designed in a way that makes it easy to extend the solution with new functionality.

¹The Calvin repository can be downloaded from <https://github.com/EricssonResearch/calvin-base>.

References

- [1] L. Atzori et al., *The Internet of Things: A survey*, Computer networks, Volume 54, Issue 15, pp. 2787-2805, 2010.
- [2] M. Armbrust et al., *Above the Clouds: A Berkeley View of Cloud Computing*, Tech. Rep No. UCB/EECS-2009-28, University of California at Berkeley, 2009
- [3] P. Persson, O. Angelsmark, *Calvin – Merging Cloud and IoT*, Procedia Computer Science, Volume 52, pp. 210-217, 2015.
- [4] S. Gerdes et al., *An architecture for authorization in constrained environments*, Internet-Draft, ACE Working Group, IETF, 2015.
<https://tools.ietf.org/pdf/draft-ietf-ace-actors-02.pdf>.
- [5] L. Seitz et al., *Authorization for the Internet of Things using OAuth 2.0*, Internet-Draft, ACE Working Group, IETF, 2016.
<https://tools.ietf.org/pdf/draft-ietf-ace-oauth-authz-01.pdf>.
- [6] L. Seitz et al., *Authorization Framework for the Internet-of-Things*, World of Wireless, Mobile and Multimedia Networks (WoWMoM), IEEE, pp. 1-6, 2013.
- [7] *Authentication, Authorization, and Access Control*, Apache, 2010.
<http://httpd.apache.org/docs/1.3/howto/auth.html>.
- [8] J. Andres, *The Basics of Information Security: Understanding the Fundamentals of InfoSec in Theory and Practice*, Syngress, pp. 42-44, 2011.
- [9] V.C. Hu et al., *Guide to Attribute Based Access Control (ABAC) Definition and Considerations*, NIST Special Publication 800-162, 2014.
- [10] S. Nair, *Short introduction to Access Control - Part 1*, Axiomatics, 2013.
<http://www.axiomatics.com/blog/entry/short-introduction-to-access-control-part-1.html>.
- [11] S. Nair, *Short introduction to Access Control - Part 2*, Axiomatics, 2013.
<https://www.axiomatics.com/blog/entry/short-introduction-to-access-control-part-2.html>.
- [12] *eXtensible Access Control Markup Language (XACML) Version 3.0*, OASIS Standard, 2013.

-
- [13] S. Nair, *XACML Reference Architecture*, Axiomatics, 2013.
<http://www.axiomatics.com/blog/entry/xacml-reference-architecture.html>.
- [14] *XACML SAML Profile Version 2.0*, OASIS Committee Specification, 2014.
- [15] *JSON Profile of XACML 3.0 Version 1.0*, OASIS Committee Specification, 2014.
- [16] *REST Profile of XACML v3.0 Version 1.0*, OASIS Committee Specification, 2014.
- [17] W. Stallings, *Cryptography and Network Security: Principles and Practice*, 5th ed., Prentice Hall Press, 2010.
- [18] M. Jones et al., *JSON Web Token (JWT)*, RFC 7519, IETF, 2015.
- [19] *Introduction to JSON Web Tokens*, Auth0, 2016.
<https://jwt.io/introduction/>
- [20] M. Jones et al., *JSON Web Signature (JWS)*, RFC 7519, IETF, 2015.
- [21] J. Persson, *Open Source release of IoT app environment Calvin*, Ericsson Research Blog, 2015.
<http://www.ericsson.com/research-blog/cloud/open-source-calvin/>
- [22] O. Angelsmark, *A closer look at Calvin*, Ericsson Research Blog, 2015.
<http://www.ericsson.com/research-blog/cloud/closer-look-calvin/>
- [23] *Storage*, Calvin Wiki, Ericsson Research, Github, 2016.
<https://github.com/EricssonResearch/calvin-base/wiki/Storage>
- [24] *Capabilities & Requirements*, Calvin Wiki, Ericsson Research, Github, 2016.
<https://github.com/EricssonResearch/calvin-base/wiki/Capabilities-&Requirements>
- [25] *Application Deployment Requirement*, Calvin Wiki, Ericsson Research, Github, 2016.
<https://github.com/EricssonResearch/calvin-base/wiki/Application-Deployment-Requirement>
- [26] *Attribute Based Access Control – Executive Summary*, NIST Special Publication 1800-3a, 2015.
- [27] G. Gebel, *The Very Latest in Authorization Standards and Trends*, Axiomatics, Cloud Identity Summit, 2014.
<http://www.slideshare.net/CloudIDSummit/authorization-v-next-cis2014-gerry-gebel>.
- [28] L. Griffin et. al., *On the performance of access control policy evaluation*, IEEE International Symposium on Policies for Distributed Systems and Networks, 2012.
- [29] P. Siriwardena, *Advanced API Security: Securing APIs with OAuth 2.0, OpenID Connect, JWS, and JWE*, Apress, 2014.

Runtime Registration Example

To enable future authorization requests, a runtime must send registration information to its *Policy Decision Point* (PDP) when it is started. The registration information consists of a compact JSON representation of the runtime attributes that are entered when starting the runtime¹.

The following example is a registration sent from a runtime named `testNode`, located in Sweden and with an owner that works for Ericsson:

```
1 {  
2   "node_name.name": "testNode",  
3   "owner.organization": "com.ericsson",  
4   "address.country": "SE"  
5 }
```

If the PDP is external, the registration info is put in a signed JSON Web Token, as explained in Section 4.2.3.

¹All supported runtime attributes are listed in the `indexed_public` section on <https://github.com/EricssonResearch/calvin-base/wiki/Application-Deployment-Requirement>.

Authorization Request/Response Example

An authorization request is sent by the *Policy Enforcement Point* (PEP) runtime to a *Policy Decision Point* (PDP) to check if access should be granted to an application or an actor. The PDP responds with an authorization decision. JSON is used for both the request and the response.

If the PDP is external, the request/response is put in a signed JSON Web Token, as explained in Section 4.2.3.

B.1 Request

An authorization request contains subject attributes and the ID of the runtime as a resource attribute. For actor authorization requests, a list of required runtime functionality is also included as an action attribute.

```
1 {
2   "subject": {
3     "first_name": "Tomas",
4     "last_name": "Nilsson",
5     "actor_signer": "Ericsson"
6   },
7   "action": {
8     "requires": ["runtime", "calvinsys.events.timer"]
9   },
10  "resource": {
11    "node_id": "a77c0687-dce8-496f-8d81-571333be6116"
12  }
13 }
```


B.2 Response

An authorization response contains a decision and possibly an `obligations` section with constraints under which the authorization decision is valid.

```
1  {
2    "decision": "permit",
3    "obligations": [
4      {
5        "id": "time_range",
6        "attributes": {
7          "start_time": "09:00",
8          "end_time": "17:00"
9        }
10     }
11  ]
12 }
```

Authorization Policy Example

The *Policy Decision Point* (PDP) uses authorization policies to evaluate requests from a *Policy Enforcement Point* (PEP). JSON is used for the policies, as shown in the following example:

```
1  {
2    "id": "policy1",
3    "description": "Security policy for user 'Tomas' or 'Gustav'
4                  'Nilsson' with actor signed by 'Ericsson'",
5    "rule_combining": "permit_overrides",
6    "target": {
7      "subject": {
8        "first_name": ["Tomas", "Gustav"],
9        "last_name": "Nilsson",
10       "actor_signer": "Ericsson"
11     }
12   },
13   "rules": [
14     {
15       "id": "policy1_rule1",
16       "description": "Permit access to 'calvinsys.events.timer',
17                   'calvinsys.io.*' and 'runtime' between
18                   09:00 and 17:00 if condition is true.",
19       "effect": "permit",
20       "target": {
21         "action": {
22           "requires": ["calvinsys.events.timer",
23                     "calvinsys.io.*", "runtime"]
24         }
25       },
26       "condition": {
27         "function": "and",
28         "attributes": [
29           {
30             "function": "equal",
31             "attributes": ["attr:resource:address.country",
32                          ["SE", "DK"]]
33           },
34       ]
35     }
36   ]
37 }
```

```
35         "function": "greater_than_or_equal",
36         "attributes": ["attr:environment:current_date",
37                       "2016-03-04"]
38     }
39 ]
40 },
41 "obligations": [
42     {
43         "id": "time_range",
44         "attributes": {
45             "start_time": "09:00",
46             "end_time": "17:00"
47         }
48     }
49 ]
50 }
51 ]
52 }
```



LUND
UNIVERSITY

Series of Master's theses
Department of Electrical and Information Technology
LU/LTH-EIT 2016-511

<http://www.eit.lth.se>