Master's Thesis[1]

OVERRIDING OF ACCESS CONTROL IN XACML

By

Ja'far S. Al-Qatawna

Submitted in partial fulfillment of the requirements for
The Master degree in
Information and Communication Systems Security
At the Royal Institute of Technology

Sweden, 2006

SecLab
Department of Computer and Systems Sciences (DSV)

Security, Policy and Trust Laboratory
SICS - Sweden

[1] This thesis corresponds to 20 weeks of full-time work.

Abstract

In a networked environment, information needs to be protected, therefore authorization and access control systems have been studied in the field of computer security for a long time, as a result of that, many access control mechanisms have been developed. Most of these mechanisms focused on how to define users' rights in a precise way to prevent any violation for the access control policy. To some degree classical access control models are not flexible; they either permit access or deny it completely. The access control decision is made based on the assumption that all access needs are known in advance, and these needs can be expressed by machine readable code. In many situations it's hard to predefine all the access needs, or even to express them in machine readable code. One example of such situation is an emergency case which can not be predictable. A discretionary overriding of access control mechanism is one way for handling such hard to define and unanticipated situations. The override mechanism gives the subject of the access control policy the possibility to override the denied decision, given that the subject should confirm the access (on his discretion), the access will be logged for auditing, and notification will be sent for the responsible authority. Since the override mechanism covers more access needs and helps in writing complete access control policy, the goal hers is introducing this mechanism in a standard way, which will make it applicable for wide range of applications and suitable for distributed systems where a common access control language is needed. In this thesis, the discretionary overriding of access control has been introduced in the standardized framework of the eXtensible Access Control Markup Language (XACML) which gives common language for expressing access control mechanisms. XACML has been extended to support the override mechanism. The override has been introduced as XACML obligation, and since XACML lacks a defined way for combining obligations, a new obligations-combining algorithm has been proposed. The proposed solution provides a general way for combining XACML obligations that can be used to create a chain of obligations-combining algorithms; each has its own purpose and particular type of obligations. As a proof of concept, the general solution has been implemented using Sun Microsystems open source of XACML. This helps in checking if the solution gives the intended result and if it works properly with different XACML components.

Keywords: Information system security, authorization, access control policy language, policy & mechanism, XACML, Obligations, access control override.

Preface

This thesis is part of the Master programme of Information and Communication Systems Security at the Royal Institute of Technology (KTH), Sweden. The program includes two semesters taught courses, which sums to 40 credits (60 ECTS). The study period is followed by an average five to six months of thesis project work (20 credits, 30 ECTS).

This thesis has been carried out at Security, Policy and Trust Laboratory (SPOT) of the Swedish Institute of Computer Science (SICS), which is an independent non-profit research organization. The thesis is part of an ongoing project in which SPOT investigates the use of XACML as a policy language for distributed services in the highly dynamic and decentralised networks.

SecLab, Department of Computer and Systems Sciences, SU/KTH
Forum 100
S-164 40 Kista Sweden
Phone + 46 8 16 16 91
Fax + 46 8 703 90 25
http://dsv.su.se/en/

SICS, Swedish Institute of Computer Science AB
Visit, Electrum, Isafjordsgatan 22
Box 1263, 164 29 Kista, Sweden
Phone: +46 8 633 15 00
Fax +46 8 751 72 30
www.sics.se

Acknowledgment

First and foremost, I thank God for giving me the power of knowledge and the opportunity to improve my qualification.

My sincere thanks also go to the Royal Institute of Technology (KTH) and all members of SecLab for giving me a chance to continue my Master degree here in Sweden.

I'm grateful for my supervisor Erik Rissanen for his insightful suggestions, comments and great support throughout the thesis project. Special thanks must go to Dr. Babak Sadighi for giving me the opportunity to do my thesis at the Swedish Institute of Computer Science (SICS), Stockholm. I would like to thank Dr. Ludwig Seitz and all members of the Security, Policy and Trust Lab at SICS for their support and encouragement. It was a great experience to work in the environment of SICS!

I would like to thank Dr. Matei Ciobanu and Jeffy Mwakalinga at SecLab/DSV/KTH for their support and guidance.

My deepest gratitude to my parents in Jordan, for their non-stop support, love and care through out my education. Thank goes to friends and relatives in UAE.

I would like to thank all my friends for their support and encouragement during my stay in Sweden – I will always remember you all!

Last but not least, thanks to Sweden.

Stockholm, August 2006

Ja'far Al-Qatawna

Table of contents

Abstract i

Preface ii

Acknowledgment iii

Table of contents iv

1. INTRODUCTION 1

1.1 Background 1
1.2 Problem Definition 3
1.3 Research Goal 4
1.4 Research Purpose 5
1.5 Research Method 5
1.6 Audience 6
1.7 Thesis Structure 6

2. COMPUTER SECURITY & ROLE OF ACCESS CONTROL 8

2.1 Security overview 8
2.2 Access Control Model 9
2.2.1 Access Control Policy and Mechanism 10
2.2.2 Access Control design principles 10
2.2.3 Access Control and other Security Services 10
2.2.4 Access Control Types 12
2.2.5 Access Control List and Capability List 12

3. FRAMEWORK AND RELATED STANDARD 14

3.1 Traditional Access Control limitations 14
3.2 Overriding of Access Control 14
3.2.1 Discretionary Overriding of the Access Control 15
3.3 eXtensible Access Control Markup Language (XACML) 16
3.3.1 XACML features, model and components 17
3.3.2 XACML open issues 20

4. DESIGN AND PROPOSED SOLUTION 22

4.1 Introducing Discretionary Overriding of Access Control in XACML 22
4.2 Evaluation in more complex and decentralized situations 27
4.3 Custom design for placing the override mechanism within XACML 28
4.4 Generalized solution for handling the override obligation in XACML 33

5. IMPLEMENTATION AND TECHNICAL DETAILS 44
5.1 Sun XACML implementation 44
5.2 Implementation overview 44
5.2.1 Schema modification 46
5.2.2 Effects-Combining Algorithm 48
5.2.3 Obligations-Combining Algorithm 48

6. CONCLUSION AND FUTURE WORK 51
6.1 Conclusion 51
6.2 Future work 53
6.2.1 The Authority Resolution Mechanism 53
6.2.2 Delegation 53
6.2.3 Additional use cases for obligation combining algorithm 53

References 55

Appendix A-1: WithOverrideCombinAlg implementation 57

Appendix A-2: PermitOverridesEffectAlg implementation 59

CHAPTER 1

INTRODUCTION

This chapter is an introduction to the different parts of this thesis. It will introduce the first chapter which is an important part, since it gives the reader complete overview about the work and the structure of this thesis.

1.1 Background

There is a huge deployment of computer networks in all organizations activities, this leads to change the information from its classical shape as paper or even video and sound taps to a digitized form that can be stored, accessed, updated, or deleted locally or remotely from any computer machines connected to the organization network. This huge deployment introduces new risks for the organizations, Confidentiality, Availability, and Integrity of the digitized information need to be protected and new mechanisms need to be implemented to achieve this goal. In such situation there is a need for verifying the identity of the users in the network environments, this is the task of the Authentication services that guarantees that user is who he/she claims to be, one example of such service is the login mechanism using user ID and password which is the most popular one, but unfortunately not the strongest one. To reduce the risks as much as possible there is a need also for defining who is allowed to access the stored information and what type of actions he/she is allowed to perform in every specific piece of the digitized information, this is the task of the Authorization service that ensures that the network's resources only accessible by the authorized users and also it defines precisely their permissions on these resources. For the enforcement of the Authorization service different Access Control mechanisms are used.

Access Control is one of the most important security services in the Network environments, Operating Systems, and Web Services. Once the user has been authenticated to the system, the system needs to authorize the user, i.e. to determine which system resources the user is allowed to access and what set of actions, he or she is allowed to perform on those resources.

Access Control consists of policy and mechanism. The Policy is non-technical statement of what is, and what is not allowed, while the mechanism is a method, tool, or procedure for enforcing the Access Control policy [B05]. In many situations, it is hard to predefine all the access needs; this makes the access control policy incomplete, since it will not cover all the access possibilities. There are unpredictable or urgent situations where the needs for access permission are not defined in the access control policy in advance. As example to clarify that is an access control policy that allows only the primary care physician to access the medical records for a given patient, but in real life example, there are some urgent situations that may affect the patient's life and another physician needs to access his/her medical record, so how such situation can be defined by the access control policy.This may lead to a conflict between the need for legitimate access (resource availability), and the need for protection from an unauthorized access (confidentiality and integrity of the resource) [RFS04].

Rissanen [RFS04] suggests a flexible solution that is based on the idea of discretionary overriding of the access control policy. This solution requires the overriding process to be audited, and a notification needs to be sent to management authority. It differs from other solutions in that it has the notion of Authority Resolution, which is an automatic procedure that will, given information about an override and an access control policy, find who is in a position to audit and approve the override in retroactive manner.

Many of current Access Control and Authorization systems implement Access Control mechanisms in proprietary way [IBM], this makes them limited to specific

applications and can not be used for open distributed networks, in which there is a need for sharing access control related information between different autonomous domains, therefore a common language is needed to express different access control polices for such situation [LPLKS03]. eXtensible Access Control Markup Language (XACML) which standardized by OASIS [IT] promises to be a powerful and flexible policy language for heterogeneous distributed systems. XACML provides general-purpose access control policy language. It provides syntax for managing access control to resources.

With the involvement of the computers in all the organizational (medical, commercial, industrial, and educational) activities, and with the heterogeneity of the computer networks nowadays, there is a need for more flexible and trusted Access Control systems, which need to be standardized in order to make it more convenient and inter-operable. One can imagine the advantages of introducing a good access control mechanism such as the Discretionary Overriding of Access Control mechanism in a powerful and standardized framework such XACML.

1.2 Problem Definition

Since XACML promises to be a powerful and flexible policy language in heterogeneous distributed systems, the question here is how to apply the ideas of the Discretionary Overriding of Access Control within the XACML framework.

The override approach discussed earlier depends on the differentiation between what the Subject of the Access control *can do*, what is *permitted*, and what is *forbidden*. Where the intersection of *can do*, and *forbidden*, will introduce a new access possibility called “possible-with-override”. So for the access request, there are three possibilities:

1. Permit if there is permission allows a given request.
2. Discretionary override the denial if no permission exists, but the ability to override exists.

3. Deny if there is no permission only, without ability to override.

A way of integrating the "possible-with-override" in policy evaluation process of XACML is needed. In current XACML standard (2.0) there are the following Authorisation Decisions: *Permit*, *Deny*, *Indeterminate* or *NotApplicable*, and optionally a set of *obligations*.

Obligation introduced originally by Solman [S94] in which he distinguished between the Authorisation policy which defines what a user is permitted or not permitted to do, and the Obligation policy which defines what a user must or must not do. Obligation presented in the Provisional Authority [KH00] which is included in XACML as a set of obligations send in the response from Policy Decision Point (PDP) along with the authorisation decision, and need to be fulfilled by the Policy Enforcement Points (PEP). The discretionary overriding approach also requires the overrides to be logged, audited and approved retroactively by the responsible authorities. Obligation in XACML can be used in the override mechanism for sending notification or starting the process of logging, but there is no defined way in current XACML standard enables us to distinguish between the normal access and the urgent access that need special treatment.

XACML also provides combining algorithm for both rules and policies to solve conflict cases, but nothing mentioned in the specifications about obligations combining algorithm, which can be helpful in case of obligations conflict.

1.3 Research Goal

The goal of this thesis is to propose a solution that clearly defines a way for applying the discretionary overriding of access control mechanism in XACML standard. As a result of this goal an extensions for the standard specifications are going to be added and custom design for combining access control policies and obligations are going to

be introduced within XACML. Improvement and generalized solution for handling the override mechanism in XACML are expected at the end of the design stage.

1.4 Research Purpose

Applying the discretionary Overriding of access control within the XACML framework will provide a model for writing complete and flexible policies that cover more access control needs, and it will be more suitable for distributed system where a common access control language is required. The result will be standardized and generic; this means it will be useful for wide rang of applications. Since XACML is quite new standard and opened for a lot of new ideas and initiatives, this thesis will try to contribute in improving the standard and discovering how it can be utilized for better access control models in open distributed networks.

1.5 Research Method

For achieving the goal of the thesis there will be a number of stages. The first stage consists of a literature review for the historical development of the access control models as well as reading new research papers to see the current state of the art in Access Control field in general. Then the focus will be turned on different access control overriding mechanisms and designs.

At the viability study stage, an in-depth analysis of current XACML standard specifications will be conducted to find to which extend the standard can help for designing access control mechanisms such as the discretionary overriding mechanism.

Based on the previous stages a proposed solution will be introduced in the design stage, where the following tasks are going to be accomplished:

1. Introducing Discretionary Overriding of Access Control in XACML.
2. Introducing Discretionary Overriding of Access Control in more complex and decentralized situations.
3. Custom design for placing the override mechanism within XACML.

4. Generalized the solution for handling the override as obligation.
5. Suggestion and improvement for handling obligations in XACML standard specifications.

At the implementation stage more technical details and proof of concept will be provided to check the feasibility of the design.
At the last stage a conclusion of the result will be presented as well as recommendations and future work.

1.6 Audience

This thesis is in the area of Computer Security and the reader expected to have the knowledge in this field. However some background is added about Authorization system and Access Control models for those who don’t have such specific knowledge about Access Control systems.

1.7 Thesis Structure

This section will give brief description about rest of the thesis.

Chapter 2 Computer Security & Role of Access Control:
It will discuss the Access Control model and its component. It will provide complete picture for the Authorization system and its relation with other security services to provide information Confidentiality, Integrity, and Availability.
Chapter 3 Framework and related standard:
It will discuss the idea of discretionary overriding of access control and the details of XACML standard as a framework for this thesis.
Chapter 4 Design and proposed solution:
It will show how the problem which is defined in the thesis are going to be solved based on number of use cases, custom design, generalized solution, and extensions for the standard.

Chapter 5 Implementation and Technical details:

It will provide proof of concept, guidelines, and technical details for implementing the proposed solution.

Chapter 6 Conclusion and future work:

It will conclude the result and provide some recommendation and future work.

CHAPTER 2

COMPUTER SECURITY & ROLE OF ACCESS CONTROL

2.1 Security overview

Computer Security is hard to be defined, since the meaning defers from context to context. One may define security in term of Confidentiality, Integrity and Availability (CIA). A secure system is the one which ensure these three services. However, there is trade-off between CIA services. Military fields usually focus on providing the Confidentiality service more than the other services, where the secrecy of the information is very important, so it needs to be protected during the transmission over the network. In contrary the e-commerce services focus more on providing Integrity and Availability services, where the transactions related information should be authentic and not tampered with in any way, and the service should available whenever it is needed.

Security can be compared to reliability since they have a lot in common. Reliability concerns about the system robustness in case of malefactions and failures. Similarly, security concerns about how a system deals with security policy violation in term of prevention, detection and recovery. Security is a subset of reliability, in which the security policy is part of the definition of "robust" that is applied to particular system [JVGM].

You can think about security in term of prevention, detection and recovery. A holistic view for information system security provide *protection* and *assurance* [KB00], see figure 2.1.

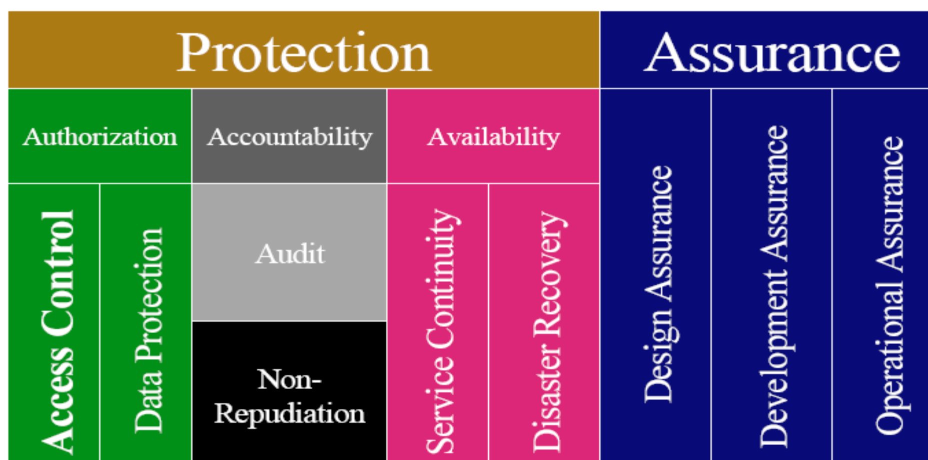


Figure 2.1: Concept of Computer Security [KB00]

Protection provides the necessary mechanisms for authorization, accountability and availability. System resources are protected by the means of access control and data protection. Auditing and non-repudiation mechanisms are use as deterrent mechanisms to provide accountability. Redundancy, backups, and faults tolerance techniques are examples of mechanisms to protect system availability. The assurance part guarantees that the security will involve from the early stage of the system design, and it will continue in the development and operational stage of the system lifetime.

2.2 Access Control Model

Access Control is one of the most important security services in the Network environments, Operating Systems, and Web Services. Once the user has been authenticated to the system, the system needs to authorize the user, i.e. to determine which system resources the user is allowed to access and what set or actions, he or she is allowed to perform on those resources. Most Access Control models have several entities: Subject (user or process) that has Right (read, write…etc) on Object (system resource, such file, directory, or service).

2.2.1 Access Control Policy and Mechanism

Access Control consists of policy and mechanism. The Policy is non-technical statement of what is, and what is not allowed, while the mechanism is a method, tool, or procedure for enforcing the Access Control policy [B05]. To make analogy, imagine a people who live in a building that consists of main entrance, elevator, washing room, and numbers of flats with their own doors. The access control policy that could be specified in the contract that people living in the building are allowed to use the main entrance, share the elevator and the washing room, and everyone has access only to his flat. All the doors with keys, locks, and PIN codes represent the access control mechanism to ensure that the policy specified in the contract will not be violated.

2.2.2 Access Control design principles

Saltzer and Schroeder [SS75] describe principles for the design and implementation of security mechanisms. One of these principles is the *Least Privilege*; the concept behind this principle is to define the user privileges precisely, which enables him to perform his task only without any extra privileges that could be misused. The other principle is the *Complete Mediation* which requires all access attempts to the system resources to be checked for authority. This principle is the fundamental of the protection system. The need for implementing such principles for various resources (fileservers, network services, database, and web services) has been the driving force in developing Access Control [CI01].

2.2.3 Access Control and other Security Services

Access control constraints the set of actions that a user (or process acts on behalf of the user) can do. The security of computer systems depends on Access control as well as other security services such as Authentication, Reference Monitor and Auditing [SS94] see figure 2.2.

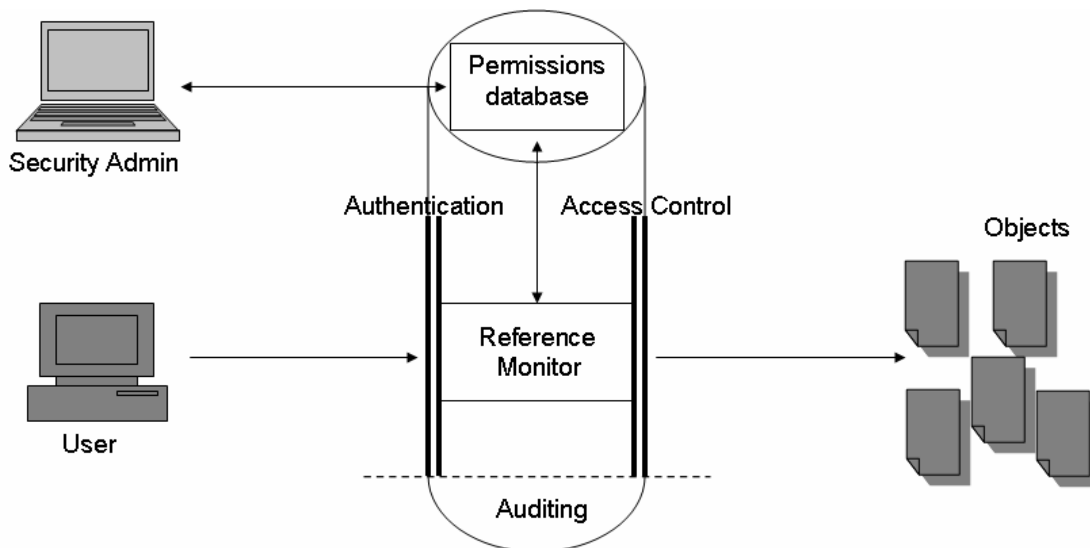


Figure 2.2: protection system [SS94]

To avoid the confusing, one need to distinguish between Authentication and Authorization. Authentication is about how to verify and ensure that the user is who he/she claims to be. The purpose of the Authentication service is to verify the identity of the user requesting access, for example by the mean of user ID and password, which is the simplest way, but unfortunately the weakest and easiest to break. On the other hand Authorization is all about Access control, where all the system resources that the legitimate user are allowed to access, are specified along with set of actions that he/she can perform on these resources. The Access control service assumes that the user has been successfully authenticated by the proper Authentication service.

Security services need to be integrated to maximize the level of protection. Based on that, Access control can be coupled with Auditing mechanism. A good Auditing mechanism allows the system to log all the relevant activities and access requests to its resources, and keep these logs for later verification. In more complex situation, it enables a real-time analysis of all access requests and generates an automatic response based on that. Auditing also helps as deterrent mechanism, if the user knows that all his/her actions on the system resource are being logged and can be verified against

and violation or malicious act, definitely this will stop him/her from thinking to misuse the granted privileges.

Reference monitor acts as enforced point for the Access control. It communicates with the authorization database to verify the user's access request against the policies stored in the authorization database and based on that it permits or denies the access.

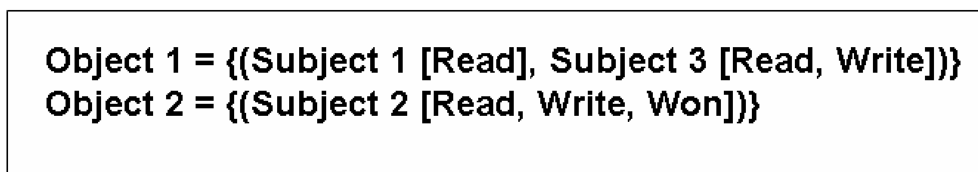
2.2.4 Access Control Types

Many approaches have been developed to design Access Control, which differ in their complexity, flexibility, and performance. In Mandatory Access Control (MAC), the policy is obligatory and the Subject is not allowed to override the policy. Under MAC, Subject is either permitted or denied to perform set of actions. On the other hand Discretionary Access Control (DAC) allows the end user (or the owner) to grant access rights on his discretion [CI01]. Most of the Operating Systems allow the user who owns a file or directory to grant read/write permissions to other users. DAC seems to be more flexible than MAC, and in some situations you could find mix of MAC and DAC implementations. Another approach for Access Control is based on the role of the Subject which is called Role Based Access Control (RBAC). RBAC is used when there is a need for accessing system resources by group of users based on their Role (System Admin, Doctor, Student…etc), in this case RBAC is more convenient and flexible since the decision is based on the role rather than on the permissions which are associated with each Subject. Some approaches are based on so called Policy or Rule Set Access Control where the access is verified against the set of rules which are predefined in the system.

2.2.5 Access Control List and Capability List

Access Control Matrix introduced by Lampson 1971, which is abstract model for representing authorization in computer systems. In matrix each subject has row, and each object has column. The intersection between any subject's row and object column represents the access mode or set of permissions for that subject [L71].

Access Control List (ACL) and Capability List are two approaches for implementing Access control Matrix [SS94]. In ACL each column from the Access Control Matrix is stored with the object it represents. A set of subjects and their permissions are associated with that object to form a row in the ACL, see figure 2.3.

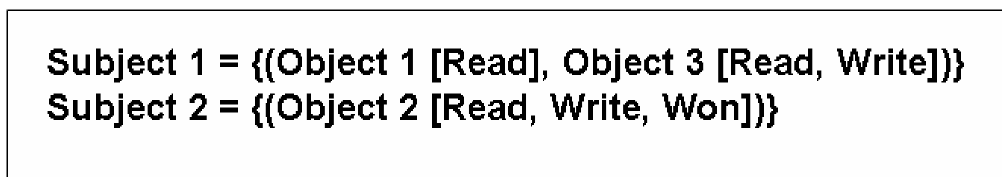


Object 1 = {(Subject 1 [Read], Subject 3 [Read, Write])}
Object 2 = {(Subject 2 [Read, Write, Won])}

Figure 2.3: Access Control List

ACL makes operations in objects easy, for example to find all the subjects who have permissions on specific object, you need only to check one row in the ACL which is related to that object. One of the disadvantages of ACL is that it becomes very hard consuming the time to operate on the Subjects of the ACL. For example if you want to delete a subject from the system you need to check all the rows of the ACL.

In Capability List each row from the Access Control Matrix is stored with the subject it represents. A set of objects and the access mode on those objects are associated to that subject, see figure 2.4.



Subject 1 = {(Object 1 [Read], Object 3 [Read, Write])}
Subject 2 = {(Object 2 [Read, Write, Won])}

Figure 2.4: Capability List

Capability List makes the operation on subject an easy one, as compared to ACL; for example finding all the permissions that a specific subject has. But in contrary it is hard to operate on object; for example to find all the subjects who have access to specific object you need to check all the rows in the Capability List.

C H A P T E R 3

FRAMEWORK AND RELATED STANDARD

3.1 Traditional Access Control limitations

In many situations, it is hard to predefine all the access needs; this makes the access control policy incomplete, since it will not cover all the access possibilities. On the other hand, there are unpredictable or urgent situations where the needs for access permission are not defined in the access control policy in advance. In Some situations, the access needs cannot be put in a machine readable form, or to be stored in the Access Control List (ACL). These situations may lead to a conflict between the need for legitimate access (resource availability), and the need for protection from unauthorized access (confidentiality and integrity of the resource) [RFS04].

3.2 Overriding of Access Control

Many researchers argue the need for overriding the access control policy in some critical or urgent situations. Povey [P00] discusses some cases such as natural disasters, urgent medical treatments, and meeting critical deadline that need optimistic security which focuses on deterrence mechanisms more than restriction mechanisms, in order to increase the flexibility and remove the gap between what the organizations need, and what the access control mechanism can do. In [LLT00] the need for an override is suggested for the medical care systems in some occasions where confidential restrictions may be overridden.

3.2.1 Discretionary Overriding of the Access Control

Classical access control models are not flexible and their policies don't cover all the access needs, based on that Rissanen [RFS04] suggests a flexible solution that is based on the idea of discretionary overriding of the access control policy. This solution requires the overriding process to be audited, and a notification needs to be sent to management authority. It differs from other solution that it has the notion of Authority Resolution, which is an automatic procedure that will, give information about an override and an access control policy, find who is in a position to audit and approve the override in retroactive manner.

The flexibility in this approach comes from the differentiation between what the Subject of the Access control *can do*, what is *permitted*, and what is *forbidden*. Where the intersection of *can do*, and *forbidden* (or denied), will introduce a new access possibility called "possible-with-override". In case a user (subject) requests access to a resource (object), we have the following possibilities:

1- Presence of permission, so the access may be performed.
2- Presence of "possible-with-override" with no permission, the access may not be performed, but can be performed if the user explicitly overrides the denied. At this stage the job of auditing and authority resolution starts.
3- If there is neither a permission nor "possible-with-override", the access cannot be performed.

In a large organization, it is impossible to have one united authority over the whole organisation, so, the solution is a decentralized approach to be responsible for audit and approval of override. In Rissanen [RFS04], approval mechanism and authority resolution are developed within the Privilege calculus [FSB01] framework, which is based on the concept of "constrained delegation". This framework distinguishes between administrative and access level permissions, and expresses all authorities in form of delegation certificates, which contain administrative right which itself contain constraints. This enables to divide up the management of access control. This division is used automatically to send notifications of override to the right people in the

organization without need for specific centralized planning for handling of the override.

The approval mechanism should be *safe;* only the legitimate authorities should be notified. It also should be unobtrusive; the most likely to understand the override should be notified without bothering the other authorities. The authorities who should be able to approve an override are precisely those who can create permission for the access that was overridden. Since the process of granting the permission is retroactive, the approvals always have precedence in case there is a conflict between multiple authorities participating in the approval mechanism. Based on all these requirements an authority resolution algorithm has been presented.

Many access control mechanisms are designed and implemented in proprietary way; this makes these mechanisms limited to specific applications and environments. Imagine the advantages of designing and implementing such mechanism in generic and standardized way, the thing that will make them suitable for open distributed systems and reusable for many applications.

This thesis will focus on how the discretionary overriding of access control can be introduced within the standardized framework of XACML to provide a flexible and standardized way for writing access control policies that cover more access needs. Before putting this mechanism in the standard framework, the next section will give an overview of XACML standard and how it can be used to express authorization systems.

3.3 eXtensible Access Control Markup Language (XACML)

Once the user had been authenticated successfully to the system, the system needs to know the set of resources that the user is allowed to access, and set of actions that he or she is allowed to perform on a given resource. eXtensible Access Control Mark up

Language (XACML) provides semantics to express policies and rules for controlling access to system resources.

XACML is general-purpose access control policy language from OASIS[1]. It provides syntax for managing access control to resources [IT]. In a large organisation access control may be managed by different units, and the access control policy may be enforced by extranet, mail server, WAN, remote-access system or any other points of enforcement, which implement access control and authorization in a proprietary manner. OASIS developed XACML standard to provide a common security policy language that can be implemented throughout the organisation and enables the organisation to manage the enforcement of all the elements of access control policy in all the components of its information systems [XAC].

3.3.1 XACML features, model and components

XACML has the following properties [IT]:

- Standardized Access Control language, which means that the language is passed through number of steps for reviewing and testing by large community of researchers, experts and users. This also enables the reuse of the authorization model in different system, and gives a common language to exchange authorization between heterogeneous systems.
- Generalization property enables XACML to express policies for any environment or any resource. The XACML policies can be used for different access control purposes with different applications.
- It is distributed. The Access control model in XACML allows the policies to be distributed in arbitrary locations; a single policy set may have reference for one or more policies in another location. XACML also provide combining algorithms to resolve the conflict between multiple policies.

[1] OASIS (Organization for the Advancement of Structured Information Standards) is a not-for-profit, international consortium that drives the development, convergence, and adoption of security and e-business standards, http://www.oasis-open.org

- Powerful. The language supports large number of data types, functions and rules, also has extensions for standards such SAML and LDAP.

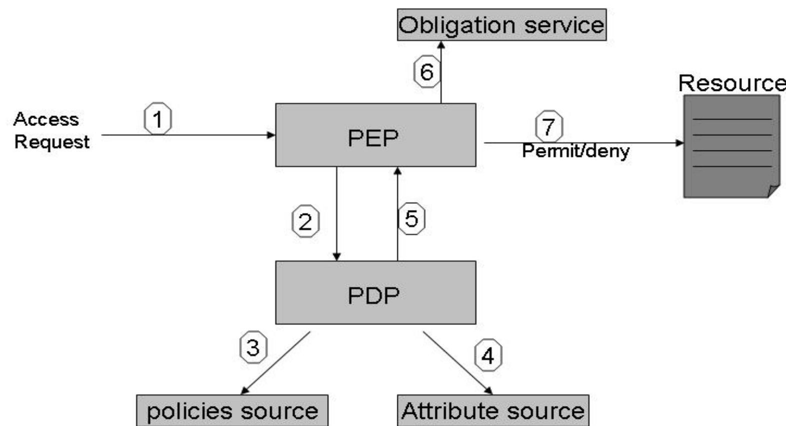


Figure 3.1 XACML typical scenario

XACML describes policy language which enables you to write *Policy* which represents single access control policy expressed through a set of *rules*. It also enables you to write *Policy Set* which is a container that holds other policies or *Policy Se*ts as well as reference to policies in a remote location. XACML describes also access control decision request/response language which allows you to form a request to ask if a specific action on some resource can be performed. The answer for the request comes as a response with one of the following values: *Permit*, *Deny*, *Undetermined* or *Not Applicable*.

In the typical scenario (see figure 3.1), the user will send access request to the Policy Enforcement Point (***PEP***), which could be a web server or a file server which protects set of resources. The ***PEP*** will form a request that contains the requester's information (Subject, Object, Action and any other related information), then it will send the decision request to the Policy Decision Point (***PDP***), which will evaluate the request against the applicable policies and return the response which contains access decision and set of obligations to the ***PEP***.

Obligation introduced originally by Solman [S94] in which he distinguished between the Authorisation policy which defined what a manager is permitted or not permitted to do, and the Obligation policy which defined what a manager must or must not do. Obligation presented in the Provisional Authority [KH00] which is included in XACML as a set of obligations (or actions) sent in the response from ***PDP*** along with the authorisation decision, and needed to be done by the ***PEP***.
Based on the decision response and the set of obligations from the ***PDP***, the ***PEP*** will decide to allow or deny the access, and it should be able to understand and discharge the set of obligations.

XACML policy language model consists of three components: *Policy Set, Policy* and *Rule*, see figure 3.2. *Rule* is the basic component of XACML *Policy*; it has *Target* which contains attributes for matching *Subject*, *Resource*, *Action*, and *Environment*, to check if the given *Rule* is applicable to specific *Request*. It has also *Effect* that may contain either *Permitted* or *Denied* value. The rule *Condition* can be used for defining Boolean expression that limits the applicability of the *rule*. Rules need to be encapsulated in a *Policy* and can not exist in isolation. *Policy* which represents single access control policy expressed through a set of *rules*, it has also *Target*, *Obligations*, and Rules combining algorithm. In *Obligation*, Policy may define operation that should be done by ***PEP*** with conjunction with the enforcement of an authorization decision; for example sending email to the owner of the information when somebody accesses his records. To enable the ***PDP*** to find which policy to be applied, *Target* is used, which is set of conditions identified by the definition of resource, subject, and action that a rule, *Policy* or *Policy Set* is intended to evaluate. *Policy Set* is a container that holds other *Policies* or *Policy Se*ts as well as reference to *Policies* in remote locations. *Policy Set* also has *Target*, *Obligations*, and Policies combining algorithms.

XACML uses a collection combining algorithms for reconciling the decisions. There are two types of combining algorithms: policy combining algorithms and rule combining algorithm. For details and examples about how to write access control policy using these XACML components can be found in [IT] and [XAC].

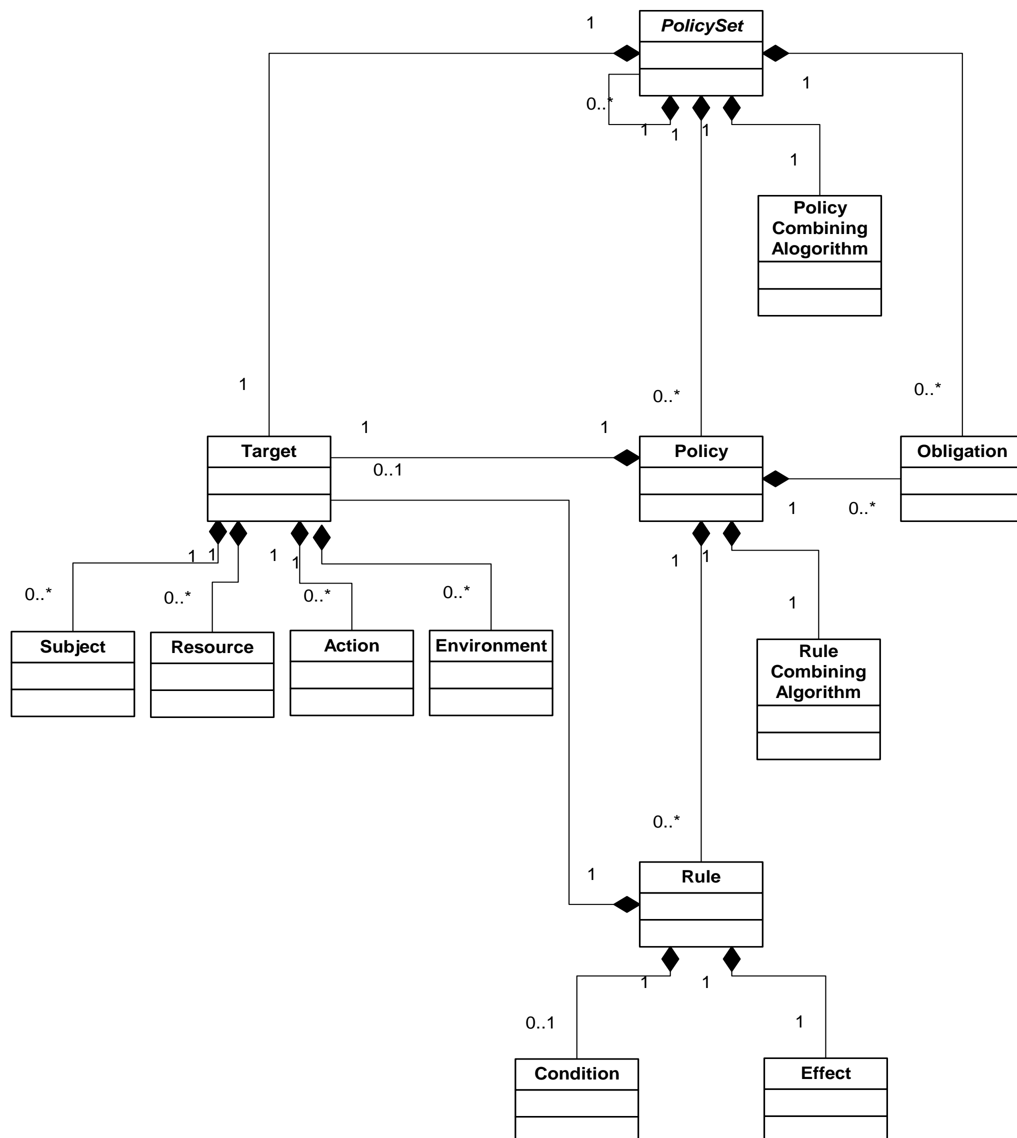


Figure 3.2 Policy language Model [XAC]

3.3.2 XACML open issues

XACML uses obligation which is an operation specified by the policy or the policy set that should be performed by the ***PEP*** in conjunction with the enforcement of an authorization decision [XAC]. Obligations are very useful since they allow adding some kind of constrains or provisional actions to the access control policy. XACML defines a way for adding obligations to the access control policy, and specifies an

evaluation process for the obligations at the policy and policy set level based on the matching of *Effect* value in the policy or policy set with the *Fulfilling* value in the obligation. Current XACML specifications (version 2.0) lack defined structure for the obligation itself; there are no defined types or categories that help the policy writer to use these obligations in generic way; the exact implementation is left open and the ***PEP*** needs to understand this implementation to be able to discharge these obligations.

XACML provides set of combining algorithms for combining rules and for combining policies, such as “Permit-override” rule-combining algorithm, which operates at the rule level and “Permit-override” policy-combining algorithm, which operates at the policy level. Policy combining algorithms are useful for solving conflict between multiple policies when each policy gives different authorization decision, but these algorithms don’t understand the obligations and XACML doesn’t provide a way for solving conflict between set of obligations or give precedence for specific obligation over another. As a result it hard to distinguish between two similar permissions, one without obligations and another with obligations, which is necessary for cases such as the access control policy override. Obligations combining algorithms need to be introducing within the XACML specifications to help handle more general access models.

XACML lacks a model for control the policy model itself. Current XACML does not give information about the source of authority and how the privileges for administrating policies can be granted or delegated, this increase the burden for managing policies in complex and distributed organisation. There is a need for a decentralized model such as the “Privilege Calculus” framework [FSB01] of which gives the ability to distinguish between administrative permissions and access control permission.

CHAPTER 4

DESIGN AND PROPOSED SOLUTION

4.1 Introducing Discretionary Overriding of Access Control in XACML using ordered policies and "First-Applicable" combining algorithm

In this section the XACML standard v2.0 example will be used as a scenario for introducing the discretionary override mechanism. The standard specifications give example of medical record stored in XML format. The medical record contains patient information and his/her primary care physician related information, see figure 4.1.

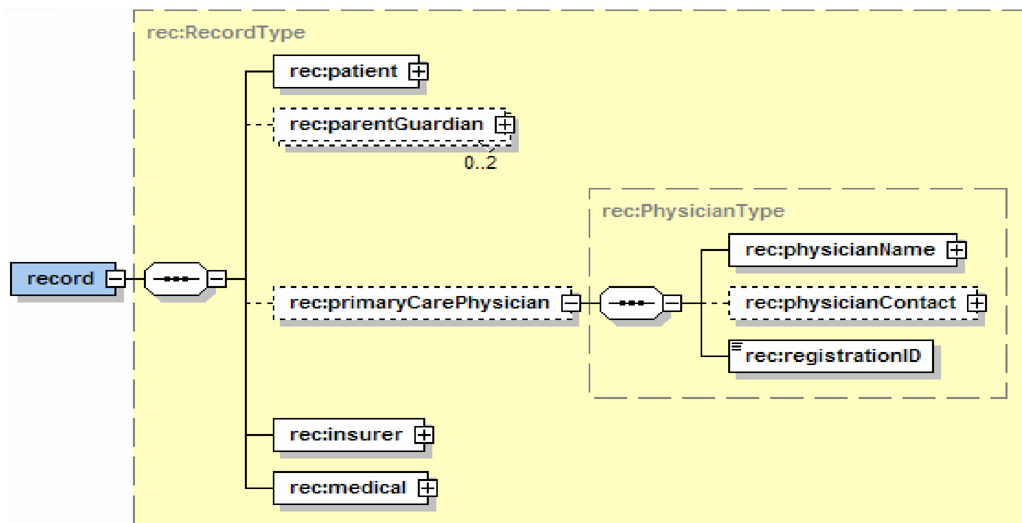


Figure 4.1: Schema for the Medical Record

If the primary care physician needs to write to his patient's medical record, an access request will be sent to the PEP that will generate the Decision Request, which

contains Subject, Resource and Action information to be sent to the PDP, see figure 4.2:

```
  <?xml version="1.0" encoding="UTF-8" ?>
- <Request...>
+ <Subject>
+ <Resource>
+ <Action>
  <Environment />
    </Request>
```

Figure 4.2: Access Request

The example defines the following Policy to protect the previous resource:
"*A physician may write any medical element in a record for which he or she is the designated primary care physician, provided an email is sent to the patient*".

To define this policy in XACML language, the policy has been structured as the following:
Access control Policy:

1. Policy header and rule combining algorithm.
2. Policy Target.
3. Policy Rule:
    a. Rule *Id* and description.
    b. Rule Target.
    c. Rule Condition.
4. Policy Obligation.

For clear picture about how the request/reopens and policy components work together see the figure 4.3.

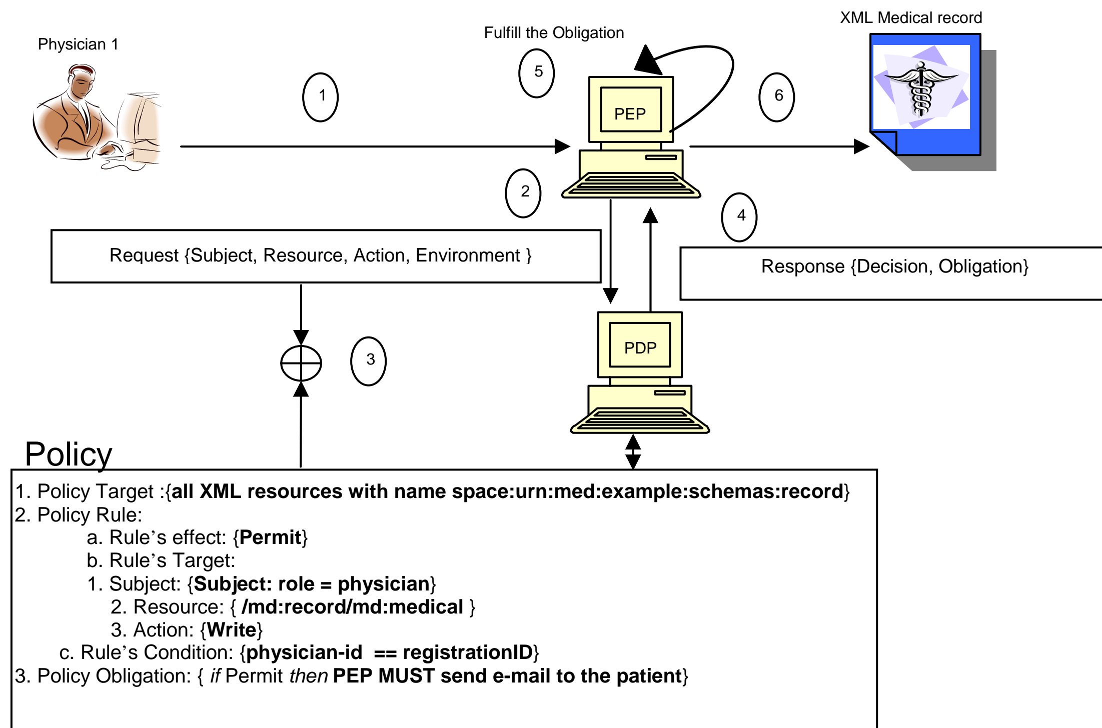


Figure 4.3 XACML Access Control example

In some situations it is hard to define a policy in a machine readable language to serve the access needs, thus the Access Control Policy will not be complete. "not complete" means that the policy does not cover all the possible access needs, where we find ourselves in a situation that we permit/deny access that we don't want to permit/deny, which could make conflict between the desired security services; Availability and Confidentiality.

The Access Control policy in the previous example is not complete. For example, in a case of emergency where the primary care physician is not available and another physician needs to access the patient's information, obviously the previous policy will deny the access and the information will not be available.

The previous policy will be restructured in a way that will enable to cover the urgent access needs that have been discussed. The way of restructuring the policy should also

enable the system to distinguish between the normal allowed accesses and the accesses in the urgent case.

Consider the following situation:

"In case of emergency where another physician (physician 2), who's not the primary care physician for the patient in this case. Physician 2 need to access the medical record for that patient, but with the policy that we created before this is not possible"

What we need:

- To solve this emergent case, we will allow physician 2 to override the previous policy given that the override will be logged and a notification will be sent to the management for approval.

- The overriding should be discretionary, i.e. the subject should explicitly override the denial, otherwise the access will be denied.

For doing that the following XACML elements will be used:

1- Obligations.
2- Policy.
3- PolicySet and policy combining algorithm, see figure 4.4.

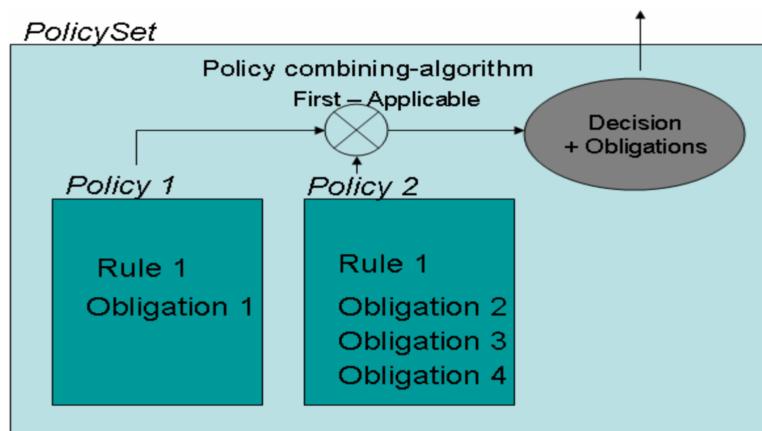


Figure 4.4: Override using ordered policies and first-applicable policy combining Algorithm

The discretionary overriding can be structured as obligation or set of obligations. In this case the following obligation can be consider:

1- In Policy 1: Obligation 1: send email to the patient if his/her record has been accessed.
2- In Policy 2 there are 3 obligations (2,3,4):
    i. override check (a way for checking explicit override from the subject)
    ii. Log the access.
    iii. Send notification to management.

Obligation 1 needs to be added also in Policy 2, since in both cases the patient should be notified. The three obligations in Policy 2 are not necessary to be separated; since they can be structured as one obligation sent from the PDP to the PEP.

The PEP should be capable to understand and discharge those obligations. Based on that we can use Deny-Biased PEP which specified in the XACML standard 2.0:

Deny-Biased PEP:

- If the ***decision*** is "Permit", then the ***PEP*** SHALL permit *access*
- If ***obligations*** accompany the ***decision***, then the ***PEP*** SHALL permit ***access*** only if it understands and it can and will discharge those ***obligations***.
- All other ***decisions*** SHALL result in the denial of ***access***.

Along with the obligations, Policy 2 will contain rule to allow physician 2 to access for example: allow subject with role = physician to access.

Since the Obligation shall be passed up to the next level of evaluation only if the effect of the policy being evaluated matches the value of the *FulfillOn* attribute of the obligation, and to preserve both the previous policy and our new policy; we can put them in one PolicySet.

Now in case there are two policies; one with normal permit and another has permit with override, the normal permit should have precedence. For achieving this purpose

the "First-applicable" combining algorithm will be used. In the case of the "First-applicable" combining algorithm, the combined result is the same as the result of evaluating the first policy element in the list of policies whose target is applicable to the decision request.

As we can see, introducing the possibility to override in this case makes the Access Control model more flexible and able to create policy which covers more access needs. But the question is whether the way that is used to introduce the override mechanism applicable to more complex cases or not, i.e. is it possible to generalize this way for using it wherever there is Access Control overriding need.

4.2 Evaluation in more complex and decentralized situations

It was possible in the previous case to introduce the override mechanism by ordering the policies and using the "First-Applicable" policy combining algorithm. The assumption for the previous proposed solution is the existence of centralized organization where all policies are administrated by single authority or domain. In this case all the policies will be predefined and located in one repository and it will be possible to determine which policy that will give the possibility to override, as in the previous case there was one PolicySet that contains all the policies where the order of these policies and which one will give the possibility to override are known in advance, so it was easy for the PDP to evaluate the policy set and combine it with the "First-Applicable" policy combining algorithm.

Managing Access Control policies in complex networks or distributed systems is handled by different autonomous administrative domains. One user may have access privileges in one domain but not in the other, and in other situations it could be convenient to give the user privileges that he/she can use them in multiple domains. However in all these cases there is a need for cooperation between these distributed domains.

XACML allows policies to be distributed in different locations and enforced by several enforcement points. However; XACML doesn't provide an explicit way for distributing policies, instead it defines two elements; *<PolicySetReference>* and *<PolicyIdReference>* , that give the possibility to write a policy or PolicySet that can refer to other policies stored in arbitrary locations. The PDP will collect the policies from these different locations and combine them for single authorization decision using set of combination algorithm.

Now, let's go back to the discretionary override mechanism, which is introduced in the previous section to see if it applicable to such distributed situation. The PDP will not be able to predict the order of policies since they are scattered in deferent locations, so it will not be possible to use the "First-Applicable" policy combining algorithm with ordered policies for such situation. The PDP will collect decisions from different policies without knowledge about which one will give the possibly to override since this possibility introduced as obligation within the policy. The PDP will not be able to give precedence for the normal permit since it can not predict the order of policies and can not distinguish between the normal permit and the one with override since the override is structured as obligation. As a conclusion the proposed solution in which is has been discussed in the previous section will serve only simple centralized situation where it is possible to order and get complete knowledge about the policies. Since we are more concerned about general solution that can cover more situations such as in distributed system we need to look for another way to design the discretionary override mechanism within XACML.

4.3 Custom design for placing the override mechanism within XACML

From the previous discussion we came out with the conclusion that XACML with its current specifications is not completely supporting the discretionary overriding of access control. As a possible way for introducing the overriding mechanism, the override is introduced as obligation. XACML doesn't give any clear way for

combining obligations, and the policy evaluation doesn't understand obligations, instead it simply collects all the obligations that match the final result. As a result of this; the PDP is unable to distinguish or give precedence between normal access and access in urgent case, i.e. the one with discretionary override which needs to be audited and notified.

One possible way for solving the problem is writing a custom combining algorithm to deal with the overriding obligation. This may not be supported by the standard, but it is kind of workaround.

XACML standard needs to be extended to provide solution that supports the discretionary overriding mechanism. The extensions are going to be new policy combining algorithm which will be able to understand obligations as well as decisions.

Let's start thinking about the characteristics of such algorithm:

- The algorithm takes list of policies as an input.
- It will return decision which may have any of the following value *{Permit, Deny, NotApplicable, Indeterminate}* with/without override obligation.
- The evaluation function of the algorithm will check the decision values and existence of overriding obligation in the policy being evaluated:
    1. If any policy is evaluated to "Permit" with no override obligation exists, the search will be terminated and the final result is "Permit"; this means to give the precedence to normal access.
    2. If the policy is evaluated to "Permit" with override obligation exits, the decision and the override obligation will be saved and the search will continue.
    3. If all policies are evaluated to "NotApplicable" the Decision will be "NotApplicable".
    4. If all policies are evaluated to "Deny" the Decision will be "Deny".
    5. If an error happens the Decision will be "Indeterminate", provided no other policies evaluated to "Permit" or "Deny".

Let’s take simple example to see how this will work, assume the previous scenario in which there is one PolicySet and two policies. You may consider these two policies distributed in different locations and have reference inside the PolicySet or placed direct in the PolicySet. Using the custom combining algorithm the evaluation order should not affect the final result.

```
<PolicySet PolicyCombingAlqId:"Possible-with-override">
    <Resource>MedicalRecord
    <Action>Read
<Policy poicyId:1>
     <Subject> Physician1
   <Rule Effect="Permit">
</Policy>

<Policy policyId:2>
     <Subject> "any-other-Physician"
   <Rule Effect="Permit">
   <Obligations>
        with-override FulfillOn="Permit"
   </Obligations>
</Policy>
</PolicySet>
```

Figure 4.5: Overriding with custom combining algorithm

Putting in mind the characteristics of the custom policy combining algorithm, let’s check how the PolicySet in figure 4.5 can be evaluated:

Case 1: the primary care physician (physician1) tries to read the medical record, in this case we have normal access that doesn’t need any special treatment from the PDP and PEP, but this case will be given the precedence. Now if physician1 sends request to read the medical record, based on the custom algorithm, the policy that has normal permit without obligation to override will be given the precedence and its affect will be returned as decision. Thus, if policy 1 evaluated first then we will get “Permit” without override obligation, and based on the custom algorithm no need to continue the evaluation.

Now, let's suppose policy 2 evaluated first. The policy will be applicable or not, but let say it is applicable assuming that the "Subject Role" as "Physician" is used for target matching process. So if the request matched the policy target and we got "Permit" with override obligation, the result will be saved and the evaluation will continue to next policy, which is policy 1. Since policy 1 matches the request and have "permit" without override obligation, it will return "Permit" as final result, and this the same as if the evaluation start by policy one instead of policy 2.

Case 2: it's an emergency case, where another physician needs to be given the possibility to override. If another physician who is not the primary care physician send access request, simply the request will not match policy 1 target no matter policy 1 is evaluated first or second. The request will match policy 2 target and the result will be saved and since no other policies need to be evaluated the combining algorithm will return "Permit" with override obligation.

As we can see, by using the custom combining algorithm the order of policies doesn't affect the evaluation process, even we don't need to know how the policy need to be ordered, and it is possible for the PDP to distinguish between the normal and urgent accesses.
Figure 4.6 shows a pseudo-code representing the evaluation strategy of the custom policy combining algorithm.

```
Decision With-Override-PolicyCombiningAlgorithm(Policy policy[])
{
       Boolean atLeastOneError    = false;
       Boolean atLeastOneDeny     = false;
       Boolean atLeastOneOverride = false;
       for( i=0 ; i < lengthOf(policy) ; i++ )
       {
              Decision decision = evaluate(policy[i]);
              if (decision == Deny)
              {
                     atLeastOneDeny = true;
                     continue;
              }
              if (decision == Permit)
              {
                      if (with-override obligation)
                      {
                            atLeatOneOveride = true;
                            continue;
                      }
                      else return Permit;

              }
              if (decision == NotApplicable)
              {
                     continue;
              }
              if (decision == Indeterminate)
              {
                     atLeastOneError = true;
                     continue;
              }
       }
       if (atLeastOneOverride)
       {
              return Decision (Permit, with-override obligation);
       if (atLeastOneDeny)
       {
              return Deny;
       }
       if (atLeastOneError)
       {
              return Indeterminate;
       }
       return NotApplicable;
}
```

Figure 4.6: pseudo-code for the custom policy combining algorithm:

4.4 Generalized solution for handling the override obligation in XACML

Michiharu [K05] tried to solve the problem that there is no definition in XACML about concrete obligation. Another problem he tried to solve is combining obligations since there is no clear way specified in XACML for combining obligations. The solution suggests two proposals, obligation categories and obligation combining algorithm.

As a way for providing sets of useful obligations in XACML, the proposed categories define semantics of typical obligations. Some obligations need to be performed sequentially after the action, for example, a physician may start reading some medical record as an action and after that an e-mail will be sent to the patient as first obligation, then another e-mail will be sent to the primary care physician as second obligation. Some cases required information to be processed before allowing the action, for example if personnel need to read some information from the customers' database, sometimes private information need to be encrypted before allowing the access, this type of obligation called data processing. The proposal suggests the following obligation categories:

- Atomic
- Sequential
- Asynchronous
- Supplemental
- Data-processing

For multiple obligations in the same category the solution suggests obligation combining algorithm which combines obligations per obligations categories. The algorithm preserves the obligation sequence and doesn't provide conflict resolution. Bill [BIL] proposed that to combine obligations, first the obligation categories must be combined, then combine members of each category. The proposal also suggests exclusive sequence on the member of the same category to define precedence.

The above proposals are useful in case there is a need to have categories for obligations and there is a need to have precedence within the members of the same category. With these proposals you can not give precedence for policy based on the existence of some obligations or not; since the proposals don't consider the interaction between the policy evaluation and the obligation evaluation. This solution doesn't solve the problem of having two policies, one without obligation and one with obligation and you need to give the precedence -at the policy level- to the policy without obligation, as in the discretionary overriding mechanism, where there is a need for solution that distinguishes between normal access and the one which possible with override, since the possibility of overriding designed as obligation, the solution should understand both policies effects and obligations and give the precedence at the policy level.

In the previous section the problem of applying the discretionary overriding have been solved by custom combining algorithm which understands policies obligations as well as effects, as a result the algorithm was able to distinguish between the normal permitted access and the access which only permitted with obligation for discretionary overriding of access control.

What's needed at this stage is designing a general solution that gives the ability to use different combinations of policies effects combining algorithm with obligations combining algorithm, i.e. how the custom combining algorithm can be divided into two independent parts; one treats the effects of policies and one treats obligations for overriding purpose or even for other required obligations combining purposes.

For achieving the general solution, we should have two combining algorithms, one for policies effects and one for obligations. The interaction between these two algorithms should be defined in a general and precise way, which will help in decoupling them.

Let's start thinking how the evaluation process will be in these algorithms. For the first algorithm which will treat the policies effects, it will take a list of policies as an

input and it will evaluate their effects to return one final result that could be *Permit, Deny, NotApplicable,* or *Indeterminate*. This look exactly similar to the way that the standard policy combining algorithms of XACML are working, but since the evaluation process will also return set of obligations that match the resulting effect (the final decision), the algorithm should also understand the policies without obligations, this will help later with the evaluation process of the obligations combining algorithm in which there is a need for distinguishing between the normal permitted access and the one which permitted with override obligation. We will not pay much attention to the cases where the final result is *NotApplicable* or *Indeterminate*, since we will assume no obligations need to be returned in these cases. So the following results are the possible results from the effects combining algorithm:

- *Deny* with set of obligations which their *FulfillOn* values match "*Deny*" effect.
- *Permit* with/without override obligation and/or other obligations which their *FulfillOn* values match "Permit" effect.

Now, let's start thinking about the obligations that will result from the effects combining algorithms. These obligations will be input to the obligations combining algorithm. In typical situation there will be a *PolicySet* which has number of policies. Some of these policies may contain override obligations or any other obligations, and some policies may be without obligations. Note that the obligations may come from the *PolicySet* itself; these obligations don't belong to any policy and needs to be fulfilled if they match the final effect of their *PolicySet*, See figure 4.6.

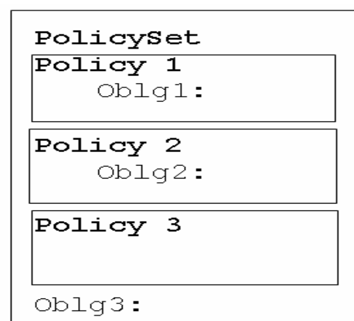


Fig 4.6: *PolicySet* with three policies and obligations

As a general result for applying the effects combining algorithm on this *PolicySet* we will get the following (by assuming that all the policies effects and obligations match the final result of the *PolicySet*):

<X, ([oblg1], [oblg2], [ ]), oblg3, [ ] >

Where X is either permit or deny. Note that oblg3 is treated as special obligation since it comes from the *PolicySet*, not from the other policies, later on this makes the further processing by the obligations combining algorithm easier. The override obligation could be any one of oblg1, and oblg2, but the existence of other set of obligations or empty set [ ] within the list of set of obligation, along with "Permit" effect, means that there are normal access and no need for overriding. These obligations also could be normal obligations that need to be fulfilled directly, or can be passed to another combining algorithm for further processing; Michiharu & Bill (XACML TC) proposals are useful for such further processing. Also note that there is possibility to have a policies without obligations, for this reason the empty set [ ] is introduced here, this will help the obligations combining algorithm to give the precedence for normal permitted access in case of the existence of two policies, one gives normal permit and another gives permit with override obligations. Additional set, which is empty at the start of evaluation, is introduced to carry the result from the obligation combining algorithm. Let's consider the following two *PolicySet*s example:

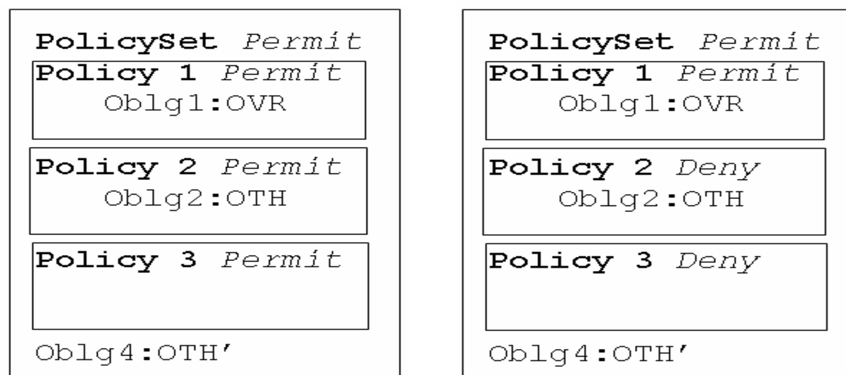


Figure 4.7: Two PolicySets one with normal access and one give access with override

In this example assume the final result of the effect combining algorithm is *Permit*. In the first *PolicySet*, policy 1 has override obligation OVR, policy 2 has other obligations OTH that could be for encryption, sending emails or whatever, and policy 3 doesn't has any obligation. The *PolicySet* itself has some other obligations OTH′. We will get the following output from the effects combining algorithm:

<Permit, ([OVR], [OTH], [ ]), OTH′, [ ]>

When these obligations are passed to the obligations combining algorithm, it should return only OTH and OTH′:

<Permit, ([OTH], [ ]), OTH′, [ ]>

Because the existence of "Permit" and empty set of obligations and/or the other set of policies obligation (OTH), this means there is at least one policy that gives "Permit" without any override obligation, so the override obligation will be dropped, and no need for overriding the access control policy in such case. In the other hand the effects combining algorithm in the second *PolicySet* will give the following result:

<Permit, ([OVR]), OTH′>

When these obligations are passed to the obligations combining algorithm, it should return OVR which will placed on the additional set and OTH′:

<Permit, ( ), OTH′, [OVR]>

In this case we have "Possible with override" access. Note that the *PolicySet* obligations are preserved also.

The evaluation strategy of the obligations-combining algorithm which is needed for the override case can be summarized as follows:

If there are permissions both with and without an obligation override obligation, then the normal access right should have priority, and there should be no special logging, prompting or sending notification as required in the override case, and the algorithm should discard any override obligation, so it will not be returned and we will get the desired effect in case we have normal access. The override algorithm should check the following cases:

1. In case there is at least one "Permit" result without an override obligation, the algorithm shall drop any override obligations.
2. In case all results have override obligations, the algorithm shall combine the obligations into a single override obligation.

See figure 4.10, which represents a pseudo-code for Obligations-combining Algorithm with example representing the evaluation strategy for the override case.

Formal specifications of the general design

The general design consists of two combining algorithms; the first one is **effects-combining algorithm** that operates at the *PolicySet* level by evaluating the entire set of policies in the *PolicySet*. The result shall contain single effect and list of set of obligations; this result forms the input for the second combining algorithm. The second algorithm is **obligations-combining algorithm** which can be a single obligations combining algorithm to achieve one purpose such the access control override, or it can be a chain of obligations combining algorithms, each has its inputs and outputs for different purposes; such as override, encryption, notification, truncation…etc.

Policy Set evaluation:

1. Match the *PolicySet* target values which is in the format: subjects, resources, actions, and environments, with the values in the request context which is also in the same format, to check the applicability of the given *PolicySet* to the incoming request. This target matching should be according to the specifications of Target evaluation in XACML standard.

2. If the target evaluate to "Match" then the value of the *PolicySet* shall be determined by the evaluation of the entire set of *Policies* and *PolicySets*, according to the **effects-combining algorithm**. In case the *PolicySet/Policy* is evaluated to *Indeterminate* or *NotApplicable*, its result will be returned without further processing since no obligations need to be returned in this case.

3. In other case:

a. Evaluate each *Policy* in the *PolicySet*. If the policy target matches the request, evaluate the rules effects according to the standard specifications (using rule-combining algorithm), and then collect the obligations of the *Policy* being evaluated that match the policy effect value in a set of obligation. If there are no obligations in the *Policy*, then the set of obligations is empty. If there are (n) policies the result of ($Policy_i$), where (i: 1 – n), will look as follow:

$$< Effect_i, ObligationSet_i >$$

b. Combine all the effects from ($\boldsymbol{Effect_1}$) to ($\boldsymbol{Effect_n}$) according to the specified **effects-combining algorithm** to get a final effect for the *PolicySet* (*Permit/Deny*); for example if the specified effect in the effects-combining algorithms is *Permit* and at least one *Policy* evaluated to *Permit*, then the final effect will be *Permit*.

4. Check if there is obligation in the *PolicySet* itself, which doesn't belong to any *Policy* and its "FulfillOn" value match the *PolicySet* final effect. It can be one obligation or set of obligations, this will be denoted as special obligation set (***OblgS***).

5. At this step all the obligations have been collected and a single final effect has been the result from the **effects-combining algorithm**. The following steps describe how these obligations are going to be combined using the **obligations-combining algorithm**:

   a. Combine all the obligations that match the final effect resulted from the previous steps according to the **obligations-combining algorithm.** Since this algorithm is intended to be generic, and can be used for different obligation combining purposes, its input and output will be specified in a general way and the specific combining function can be specified based on the required purpose as in the override example discussed in the previous section. From the previous results the input for the **obligations-combining algorithm** can be expressed as:

   <*Effect, (ObligationSet1, OblgigationSet2 …),* ***OblgS****, WorkingSet* >

   The *Effect* parameter is the final effect which resulted from the *PolicySet* evaluation according to the **effects-combining algorithm**. The list of obligation sets *(ObligationSet1, ObligationSet2 …)* is the sets of obligations that resulted from each individual policy evaluation. Each set may contain number of obligations that need to be processed by the combining algorithm, or it may be an empty set (Ø) which indicates the existence of one or more policies that match the target and give their effect which don't have any obligation. ***OblgS*** is the set of special obligations that come direct from the *PolicySet*. The *WorkingSet* is additional set which will carry the specific type of obligations which will be return by the **obligations-combining algorithm**

This list of obligations can be passed to a chain of **obligations-combining algorithms**; each combining algorithm can be used for specific task and then

produce subset of obligations that can be passed to another **obligations-combining algorithm** to perform another task. As a final result of this chain of obligations combining, a set of obligations will be produced and this set is what needs to be fulfilled by the Policy Enforcement Point PEP.

b. Now as a final step, combine the set of obligations from previous step with the final effect from step (3b) to get the final decision along with set of obligations that need to be discharged by the PEP:

<Effect, Obligations>

In the following figures: 4.9 and 4.10 the previous evaluation steps for the **effects-combining algorithm** and the **obligations-combining algorithm** are represented as pseudo-code:

```
Decision PermitOverride-EffectCombiningAlg (Policy policy[])
{
       Boolean atLeastOneDeny     = false;
       Boolean atLeastOnePermit   = false;
       Boolean atLeastOneError    = false;

       List Obligation Oblg1      = null; // obligations with Deny effects
       List Obligation Oblg2      = null; // obligations with Permit effects
       Set  Obligation OblgS      = null; // obligations from the PolicySet

       for( i=0 ; i < lengthOf(policy) ; i++ )
       {
              Decision decision = evaluate(policy[i]);

              if (decision == Deny)
              {
                     atLeastOneDeny = true;
                     Oblg1.add(Policy[i].obligations);
                     Continue;
              }
              if (decision == Permit)
              {
                     atLeastOnePermit = true;
                     Oblg2.add(Policy[i].obligations);
                     Continue;
              }

              if (decision == NotApplicable)
              {
                      Continue;
              }
              if (decision == Indeterminate)
              {
                     atLeastOneError = true;
                     Continue;
              }
       }
       OblgS.add(PolicySet.obligations)

       if (atLeastOnePermit)
       {
              return Decision(Permit, Oblg2, OblgS);
       }

       if (atLeastOneDeny)
       {
              return Decision(Deny, Oblg1, OblgS);
       }
       if (atLeastOneError)
       {
              return Indeterminate;
       }
       return NotApplicable;
}
```

Figure 4.9: Pseudo-code for “Permit-override” Effects-combining Algorithm

```
Decision ObligationCombiningAlg (Int effect, List Obligations, Set OblgS,
                                 Set WorkingSet)
{

 // since you have the input on this form, you can define your own
 // obligations combining algorithm or chain of combining algorithms
 // depending on the required obligations combining purpose.

 /*=====================================================================
  * The following part of the pseudo-code is an example of one obligations
  * combining algorithm which allow to discretionary override the access
  * control policy using set of override obligations denoted as
  * "override obligation". The override obligation represents a set of
  * actions sent from the PDP and need to be discharged by the PEP. These
  * actions are: 1- prompting the user for confirmation before performing
  * the access, 2- logging the access and 3- sending notification for a
  * responsibly authority for retroactive approval.
  *=====================================================================*/

      Set    oblg1   = null;      // to carry the override obligations

      Boolean allWithOverride    =      true;

      if (Effect == Deny)
      {
            return Decision(Deny, List Obligations, Set OblgS, null);
      }
      if (Effect == Permit)
      {
            for (i=0; i < lengthOf(Obligations); i++ )
            {
                  Boolean hasOverride = false;

                  if (Obligation[i].contains("override obligation")
                  {
                        hasOverride = true;
                        oblg1.add("override obligation")
                        Obligation[i] = Oblg[i].remove
                                          ("override obligation");
                        Continue;
                  }
                  if (hasOverride = false)
                  {
                        allWithOverride = false;
                  }
            }
      if (allWithOverride = true)
      {
            WorkingSet.add(oblg1);
            return Decision(Permit, Obligation, OblgS, WorkingSet);
      }
      if (allWithOverride = false)
      {
            return Decision(Permit, Obligation, OblgS, WorkingSet);
      }
    }
}
```

Figure 4.10: Pseudo-code for Obligations-combining Algorithm with override example

C H A P T E R 5

IMPLEMENTATION AND TECHNICAL DETAILS

This chapter explains the implementation of the proposed solution which has been described in the previous chapter. The implementation aims to provide a proof of concept in order to demonstrate the feasibility of the proposed solution, and to test the design and show that it can be applicable for real applications. For this purpose the solution has been implemented based on Sun XACML open source implementation[1].

5.1 Sun XACML implementation

This Java implementation of XACML standard was developed by Sun Microsystems, at the Internet Security Research Group as a part of an ongoing project on Internet Authorization [SUN]. The open source implementation provide support for all XACML features by providing set of APIs that covers all the mandatory functionality of XACML, such as parsing policies or requests/responses, matching targets, evaluating policy, combining algorithms…etc. The implementation also supports extending the standard by providing APIs for adding new functionality and retrieval mechanisms. For complete list of the APIs and how to use this implementation refer to [SPG].

5.2 Implementation overview

The implementation consists of general purpose implementation of Obligations-Combining Algorithm along and Effects-Combining Algorithm. The implementation contains the following classes[2]:

---

[1] This is an open source implementation of the OASIS XACML standard, written in the Java™ programming language. For more information about the project see http://sunxacml.sourceforge.net

[2] For avoiding a lot of code in the thesis, only the implementation of *PermitOverridesEffectAlg* and *WithOverrideCombinAlg* are added to the appendix.

- *abstract class ObligationCombiningAlgorithm*: General abstract class that represent the base type for all obligation combining algorithms. It has one method that must implemented and return result of class type *CompounResult*.
- *public class CompoundResult*: defines an object that represents the input and output of the Obligations-Combining Algorithm. Object of this class has the following instance variables: effect of type integer, List of set of Obligations, Set of PolicySet Obligations, and additional Set of Obligations.
- *public abstract class OblgCombAlgFactory*: Provides a factory mechanism for installing and retrieving Obligations combining algorithms.
- *public class BaseOblgCombAlgFactory extends OblgCombAlgFactory*: This is a basic implementation of OblgCombAlgFactory. It implements the insertion and retrieval methods.
- *Public class StandardOblgComAlgFactory extends BaseOblgCombAlgFactory*: provide initializer for the supported algorithms in a similar way as in the standard implementation, but in this case there is only one obligations-combining algorithm which implements the discretionary overriding of access control mechanism.
- *Public class WithOverrideCombinAlg extends ObligationCombiningAlgorithm*: an implementation of discretionary overriding of access control mechanism based on the pseudo-code in figure 4.10 in chapter 4.
- *public interface OblgCombAlgFactoryProxy*: A simple proxy interface used to install new OblgCombAlgFactory.
- *public class PermitOverridesEffectAlg extends PolicyCombiningAlgorithm*: new policy combining algorithm based on the pseudo-code in figure 4.9 in chapter 4, which handles effects and understand obligations to prepare the input for the obligations-combing algorithm.
- *public class NewFactoryProxy implements CombiningAlgFactoryProxy*: to install new *CombiningAlgFactory* and add the *PermitOverridesEffectAlg.*

For testing these two algorithms a PDP needs to be implemented, so we can feed this PDP with XACML Request and XACML Policy, which contains PolicySet as a root policy and other policies contained in this PolicySet, then we can change the effects and the contained obligations in these policies to test the behaviour of the new combining algorithms with different input.

Some modification has been made for the existing implementation to support the new Obligations-combining algorithm since its input has argument contains a set of obligations come from the PolicySet. In order to get access to these obligations the argument of the *combine( )* function which used by all the standard combining algorithms has been modified by adding new parameter, which represents a PolicySet Object, and this gives the effects-combining algorithm access to the PolicySet obligations, so it can prepare the input for the obligations-combining algorithm.

5.2.1 Schema modification

Since XACML policy is written in XML syntax, the standard uses XML schema definition language (XSD) to define and describe the structure of the XACML policy. XML schema is powerful and extensible; it defines the elements and attributes that can appear in the policy, support data types and namespaces, and defines child elements and their order in the policy [W3C].

In order to use the proposed obligations-combining algorithm some modifications has been made to the policy schema which used the PDP to validate the XACML policy. To enable the policy to use this algorithm additional element is defined in the policy schema which represent the identifier for the obligations-combining algorithm that need to be used in the policy, see figure 5.1. This allows us to add a specific obligations-combining algorithm as element to the XACML policy and parsing it as DOM element[3]. Since the Maximum occurrence vale for this element is unbounded,

[3] XML Document Object Model (DOM). The DOM presents an XML document as a tree structure, and gives access to the structure through a set of objects. http://www.w3schools.com/dom/default.asp

you can add more than one obligations-combining algorithm if you have a chain of combining tasks with different algorithms. Each one should have specific *Id*.

```
<!-- -->
<xs:element name="PolicySet" type="xacml:PolicySetType"/>
<xs:complexType name="PolicySetType">
        <xs:sequence>
                <xs:element ref="xacml:Description" minOccurs="0"/>
                <xs:element ref="xacml:PolicyIssuer" minOccurs="0"/>
                <xs:element ref="xacml:PolicySetDefaults" minOccurs="0"/>
                <xs:element ref="xacml:OblgCombAlg" minOccurs="0" maxOccurs="unbounded"/>
                <xs:element ref="xacml:Target"/>
                                .
                                .
                                .
</xs:complexType>
<!-- -->
        .
        .
        .
<!-- -->
<xs:element name="OblgCombAlg" type="xacml:OblgCombAlgType"/>
<xs:complexType name="OblgCombAlgType">
        <xs:attribute name="OblgCombAlgId" type="xs:anyURI" use="required"/>
</xs:complexType>
<!-- -->
        .
        .
        .
```

Figure 5.1: Policy Schema modification

```
<?xml version="1.0" encoding="UTF-8"?>
<PolicySet PolicySetId="PolicySet1" xmlns="urn:oasis:names:tc:xacml:3.0:policy:schema:wd-11"
PolicyCombiningAlgId="http://www.sics.se/spot/xacml:effect-combining-algorithm:permit-effect-overrides">
<OblgCombAlg OblgCombAlgId = "http://www.sics.se/spot/xacml/Possible_With_Override"/>
                        .
                        .
                        .
</PolicySet>
```

Figure 5.2: XACML PolicySet with the two proposed algorithms

As you can see from figure 5.2, the effects-combining algorithm has been added as policy-combining algorithm since in general they are the same the only deference is that the effects-combining algorithm is able to understand the obligation and prepare

the input to the obligations-combining algorithm, thus no need for any modification for the Schema to support this algorithm.

5.2.2 Effects-Combining Algorithm

The effects-combining algorithm operates at the PolicySet level by evaluating the entire policies in a given PolicySet. The input for this algorithm is list of policies and it returns the final result that include single effect and/or set of obligations to be included in the XACML response in a similar way as the other standardized policy-combining algorithms. The special thing with this effects-combining algorithm is its entire functionality, since it understands obligations and prepares the input for the obligations-combining algorithms. The effects-combining algorithm calls each obligations-combing algorithm which is included in the PolicySet, and then it receives the obligations combining result and passes them to another obligations-combining algorithm if there is any, until it receives the final obligations combining result which can be included with the final result to be returned along with the evaluation decision as an XACML response.

One version of the effects-combining algorithm which has been implemented is *effect-combining-algorithm:permit-effect-overrides.* The algorithm is implemented under the (com.sun.xacml.combine) package of the sun XACML implementation. In general the algorithm shall return permit decision if at lease on policy is evaluated to permit. Before returning the final result, it checks if there are obligations-combining algorithms need to be invoked, so it will prepare the input for these obligations-combining algorithms, and then receive their combining result to include it with the final result. For the details of this algorithm implementation refer to appendix (A-2).

5.2.3 Obligations-Combining Algorithm

In order to make the obligations-combining algorithm generic, it has been implemented as abstract class, this gives the ability to define input and output for this algorithm in general way. The input/output is defined as object which has all the parameters needed for the obligations combining purpose as defined in the algorithm

design section. These parameters are used to instantiate the instant variables of the class and can be used by the abstract method of the class, which also defined in general way and needs to be implemented by any obligations-combining algorithm which inherits from this abstract class.

For applying the override mechanism, the *obligations-combining-algorithm: Possible_With_Override* has been implemented based on the pseudo-code in figure 4.10 in the design chapter. The algorithm uses an obligation to represent override actions as specified in the discretionary override mechanism which is discussed in chapter 3. If this override obligation is returned by the algorithm, it gives the possibility for overriding the denied decisions in some urgent cases. For a given request, the algorithm will search if any policy is evaluated to *Permit* without override obligation, in this case the decision will be *Permit*, and if there are any other obligations need to b fulfilled, they will be returned as well. If at least one policy is evaluated to *Permit* with override obligation and all the other policies are evaluated to *Deny*, the policy with the override obligation will take the precedence and the decision will be *Permit* with override obligation that needs to be fulfilled by the PEP, otherwise the final decision will be *Deny*. For details of this algorithm implementation refer to appendix (A-1).

This algorithm and the one for permit-effect override which discussed in the previous section have been tested with sample XACML PolicySet as shown in Figure 5.2. The PolicySet has three policies and obligations for each policy. The PolicySet also has its own obligations that need to be fulfilled in case they match the final decision. The override obligation has an Identifier (*Id*) to distinguish it from any other obligation, see figure 5.3.

```
<Obligation ObligationId="Possible_With_Override" FulfillOn="Permit">
            .
            .
            .
</Obligation>
```

Figure 5.3: Adding override obligation into XACML policy

By running these two algorithms using the implemented PDP and setting different effect values for the 3 policies which are contained in the sample PolicySet it was possible to test if the implementations give the intended result.

CHAPTER 6

CONCLUSION AND FUTURE WORK

6.1 Conclusion

The goal of this thesis was to provide a solution that clearly defines a way for introducing the discretionary overriding of access control in XACML standard. As a first attempt to introduce the override mechanism (the discretionary overriding of access control); XACML has been used with its current specification (XACML v2.0) without any extension or modification. XACML Obligations have been chosen to represent the override actions; an override obligation contains three actions that need to be fulfilled by the PEP: prompting the user for confirmation before performing the access, sending notification for the responsible authority for retroactive approval and logging the access. It has been possible to apply the mechanism using obligation to represent the override with ordered policies and "first-applicable" policy-combining algorithm. By evaluating this approach in more complex and distributed situation, it didn't work, since the order of policies can not be predicted in advance. A conclusion has been drawn from the first attempt that, XACML should be extended to support the override mechanism. The extension has been introduced as custom policy combining algorithm, which is able to understand particular types of obligations such as the override obligation, additional to its ability to handle policies' effects. The custom algorithm was a "Hackish" way for solving the problem, since it just concerned about the override obligations without taking care of other obligations that might be important for further processing, in fact it wasn't a clean way for handling the override mechanism using obligation in XACML.

For solving the problem a general solution has been developed which is divided in two parts: one for handling effects (effects-combining algorithm), which is similar to the standard policy-combining algorithm but with some changes to its interface. And another algorithm, which is an extension point for handling obligations (obligations-combining algorithm). The interaction between these two algorithms has been defined in a general and precise manner. This enabled us to define input and output for these algorithms in general way and to use different combinations of effects-combining algorithm and obligations-combining algorithm. By defining the input and output of the general obligations-combining algorithm, it has been possible to create a chain of obligations combining algorithms; each has it own purpose and particular types of obligations. This helped to introduce the override mechanism as obligation while supporting other obligations that need further processing by different algorithms. One particular obligation-combining algorithm has been written to combine the override obligations, which is able to distinguish between the normal permitted access and the one which is possible with override obligation that represents an emergency use case.

As a proof of concept, the proposed solution has been implemented using the open source of Sun XACML implementation. This gave the possibility to check how the solutions can work with different components from XACML standard, such as PDP, policies, request and response. The implementation also gave the chance to test the new proposed combining algorithms and to check their output by feeding the algorithms different samples policies.

By applying the discretionary overriding of access control within XACML standard, a flexible and complete model for writing access control policy has been achieved. The model covers both normal access needs and emergent access needs. XACML lacks any defined way for combining obligation, the proposed solution provides general way for handling particular types of obligations as override obligations, and can be used for implementing other obligations combining task. Since the solution uses a standard framework; it’s applicable to a wide rang of applications, and it’s suitable for distributed systems where a common access control language is required.

6.2 Future work

This section will discuss the further research work which can be built on the work and the results that have been achieved in this thesis.

6.2.1 The Authority Resolution Mechanism

The original idea of discretionary overriding of access control has the notion of *Authority Resolution,* which is introduced later on with the override mechanism in the privilege calculus framework [RFS04] as *Authority Resolution Algorithm*. Since the override actions need to be logged and a notification message needs to be sent to the responsible authority for approval, the *Authority Resolution* provides a decentralized approach to search for appropriate authority for a given override. Along with the override mechanism, it's quite interesting to see how this *Authority Resolution* can be implemented within XACML standard.

6.2.2 Delegation

XACML version 2.0 does not give information about the policy issuer and how to delegate administration privileges to control the policy and to create permissions. Parallel to this project, researchers at SICS[1] have been working on implementation for XACML 3.0 draft which provides delegation features. It could be useful to find how the proposed solution in this thesis may work with delegation and get the benefits of the delegation features which introduced in XACML 3.0.

6.2.3 Additional use cases for obligation combining algorithm

In this thesis a general solution has been proposed which handles the override obligation when access permission is required in urgent use case. The obligations combining algorithm is designed in general way, in order to support more obligation

[1] At the Swedish Institute of Computer Science SICS, where this thesis has been carried out, Security, Policy & Trust Lab participates in the OASIS XACML technical committee to define the new version of the XACML standard, in order to bring delegation features to XACML.
http://www.sics.se/spot/xacml_3_0.html

combining purposes other than the override mechanism. So it will be possible to create a chain of obligation combining algorithms. One of the interests is to find other use case and create multilayer of obligation combining tasks, not only override case.

References

[B05] Matt Bishop, Computer Security: Art and Science, Addison Wesley, 2005, 0-321-24744-2

[BIL] Proposals and ideas for handling Obligation combination, XACML 3.0 WIKI page, http://wiki.oasis-open.org/xacml/DiscussionOnObligations 2006-08-09

[CI01] Camelot Information Technology Ltd, http://camelot.com, Differentiating between Access control Terms, Copyright 2001.

[FSB01] B. Sadighi Firozabadi, M. Sergot, and O. Bandmann, (2001). Using Authority Certificates to Create Management Structures. In Proceedings of Security Protocols, 9th International Workshop, Cambridge, UK, pages 134–145. Springer Verlag.

[IBM] XML Security: Control information access with XACML. The objectives, architecture, and basic concepts of eXtensible Access Control Markup Language. http://www-128.ibm.com/developerworks/xml/library/x-xacml/ 2006-05-30

[IT] A Brief Introduction to XACML, http://www.oasis-open.org/committees/download.php/2713/Brief_Introduction_to_XACML.html 2006-05-30

[JVGM] John Viega, Gary McGraw: Building Secure Software - how to avoid security problems, Addison-Wesly.

[K05] Michiharu, Kudo, Proposal of improved obligation for broader use cases. XACML mailing list, http://lists.oasis-open.org/archives/xacml/200510/msg00001.html 2006-08-09

[KB00] Konstantin Beznosov: Engineering Access Control For Distributed Enterprise Applications, PhD dissertation 2000.

[KH00] M. Kudo and S. Hada, XML document security based on provisional authorization, Proceedings of the Seventh ACM Conference on Computer and Communications Security, Nov 2000.

[L71] Butler W. Lampson, Protection; Proceedings of the 5th Princeton Conference on Information Sciences and Systems, Princeton, 1971.

[LLT00] J. J. Longstaff, M. A. Lockyer, M. G. Thick. 2000. A model of accountability, confidentiality and override for healthcare and other applications. Proceedings of the fifth ACM workshop on Role-based access control. ACM.

[LPLKS03] M Lorch, S Proctor, R Lepro, D Kafura, S Shah, First experiences using XACML for access control in distributed systems, Proceedings of the 2003 ACM workshop on XML security, 2003.

[RFS04] E. Rissanen, B. Sadighi Firozabadi, and M. Sergot. Discretionary overriding of access control in the privilege calculus. Proceedings of Formal Aspects in Security and Trust, Toulouse, France, 2004.

[P00] Dean Povey, Optimistic security: a new access control paradigm. In Proceedings of the 1999 workshop on New security paradigms. ACMPress. 2000.

[S94] M. Sloman, Policy Driven Management for Distributed Systems. Journal of Network and Systems Management, Volume 2, part 4. Plenum Press. 1994.

[SPG] Sun's XACML Implementation, Programmer's Guide for Version 1.2 http://sunxacml.sourceforge.net/guide.html 2006-08-01.

[SS75] J. Saltzer and M. Schroeder, "The Protection of Information in Computer Systems," Proceedings of the IEEE 63 (9), pp. 1278–1308 (Sep. 1975).

[SS94] S. Ravi Sandhu and P. Samarati, Access Control: Principles and Practice, IEEE Communication Magazine September 1994.

[SUN] Sun's XACML Implementation http://sunxacml.sourceforge.net/ 2006-08-01

[XAC] eXtensible Access Control Markup Language, (XACML) Version 2.0 OASIS Standard, 1 Feb 2005. http://docs.oasis-open.org/xacml/2.0/access_control-xacml-2.0-core-spec-os.pdf 2006-06-15.

[W3C] W3C XML Schema http://www.w3schools.com/schema/default.asp 2006-08-02

Appendix A-1: WithOverrideCombinAlg implementation

```java
package com.sun.xacml.combine;

import com.sun.xacml.Obligation;
import java.net.URI;
import java.util.HashSet;
import java.util.Iterator;
import java.util.LinkedList;
import java.util.List;
import java.util.Set;



public class   WithOverrideCombinAlg
extends ObligationCombiningAlgorithm {

	public static final String algId =
		"http://www.sics.se/spot/xacml/Possible_With_Override";

	// a URI form of the identifier
	private static URI identifierURI;

	// exception if the URI was invalid, which should never be a problem
	private static RuntimeException earlyException;

	static {
		try {
			identifierURI = URI.create(algId);
		} catch (IllegalArgumentException e) {
			earlyException = e;
		}
	}
	public WithOverrideCombinAlg() {
		super(identifierURI);

		if (earlyException != null) {
			throw earlyException;
		}
	}

	// constructor
	WithOverrideCombinAlg (URI identifier){
		super(identifier);

	}
	public CompoundResult combine(CompoundResult compoundResult){

		//for collecting override obligations
		Set oblg1 = new HashSet();
		//set to carry the returned override obligation if it is override case
		Set workingSet = new HashSet();
		boolean allWithOverrideObligation = true;

		// Override obligation Id.
		URI overrideId = URI.create("Possible_With_Override");
```

```
        int effect = compoundResult.getEffect();
        List obligations = new LinkedList(compoundResult.getObligations());
        Set policySetObligation = new HashSet();
        policySetObligation.addAll(compoundResult.getPolicySetObligation());

        if (effect == 1 ){
              return compoundResult;
        }
        if (effect == 0){
              if (obligations != null){
                    Iterator it1 = obligations.iterator();
                    while (it1.hasNext()){
                          Set obj = (Set)it1.next();
                          Iterator it2 = obj.iterator();
                          boolean hasOverride = false;
                          while (it2.hasNext()){
                                Obligation oblg =(Obligation)it2.next();
                                if (oblg.getId().equals(overrideId)){
                                      // collect the override obligation
                                      if(oblg1.size()== 0){
                                            oblg1.add(oblg);
                                      }
                                      // drop it from the original List
                                      it2.remove();
                                      hasOverride = true;
                                }
                          }
                          if (hasOverride == false){
                                allWithOverrideObligation = false;
                          }
                    }

              }

        }
        if (allWithOverrideObligation == true){
              workingSet.addAll(oblg1);
              return new CompoundResult(effect,obligations,
                          policySetObligation,workingSet);
        }
        else{
              return new CompoundResult(effect,obligations,
                          policySetObligation,workingSet);
        }
  }
```

Appendix A-2: PermitOverridesEffectAlg implementation

Note: this class is similar to the policy-combining algorithm class `"PermitOverridesPolicyAlg"` in Sun XACML implementation, but it has been modified to make it work as the effects-combining algorithm which has been introduced in this thesis.

```
package com.sun.xacml.combine;

import com.sun.xacml.AbstractPolicy;
import com.sun.xacml.EvaluationCtx;
import com.sun.xacml.MatchResult;
import com.sun.xacml.PolicySet;

import com.sun.xacml.ctx.Result;
import com.sun.xacml.ctx.Status;

import java.net.URI;
import java.util.HashSet;
import java.util.Iterator;
import java.util.LinkedList;
import java.util.List;
import java.util.Set;


/**
 * Effect-combining-algorithm:permit-effect-override
 */
public class PermitOverridesEffectAlg extends PolicyCombiningAlgorithm
{

    /**
     * The standard URN used to identify this algorithm
     */
    public static final String algId =
        "http://www.sics.se/spot/xacml:effect-combining-algorithm:" +
        "permit-effect-overrides";


    private ObligationCombiningAlgorithm oblgCombAlg;

    // a URI form of the identifier
    private static URI identifierURI;
    // exception if the URI was invalid, which should never be a problem
    private static RuntimeException earlyException;

    static {
        try {
            identifierURI = URI.create(algId);
        } catch (IllegalArgumentException e) {
            earlyException = e;
        }
    }

    /**
     * Standard constructor.
     */
    public PermitOverridesEffectAlg() {
```

```
        super(identifierURI);

        if (earlyException != null) {
            throw earlyException;
        }
    }
    /**
     * Applies the combining rule to the set of policies based on the
     * evaluation context.
     *
     * @param context the context from the request
     * @param parameters a (possibly empty) non-null <code>List</code> of
     *                   <code>CombinerParameter<code>s
     * @param policyElements the policies to combine
     * @param root policy
     * @return the result of running the combining algorithm
     */
     public Result combine(EvaluationCtx context, List parameters,
                           List policyElements, AbstractPolicy policySet) {
        boolean atLeastOneError = false;
        boolean atLeastOneDeny = false;
        boolean atLeastOnePermit = false;
        Set denyObligations = new HashSet();
        List permitObligations = new LinkedList();
        Set policySetObligations = new HashSet();
        Status firstIndeterminateStatus = null;
        Iterator it = policyElements.iterator();
        List oblgCombAlgIds = new LinkedList();
        Result result;

        // collect PolicySet obligation
        policySetObligations.addAll(policySet.getObligations());

        // collect Obligation-Combining Algorithms Ids
        oblgCombAlgIds.addAll(((PolicySet)policySet).getOblgCombIds());

        while (it.hasNext()) {
            AbstractPolicy policy =
                ((PolicyCombinerElement)(it.next())).getPolicy();

            // make sure that the policy matches the context
            MatchResult match = policy.match(context);

            if (match.getResult() == MatchResult.INDETERMINATE) {
                atLeastOneError = true;

                // keep track of the first error, regardless of cause
                if (firstIndeterminateStatus == null)
                    firstIndeterminateStatus = match.getStatus();
            } else if (match.getResult() == MatchResult.MATCH) {
                // now we evaluate the policy
                result = policy.evaluate(context);
                int effect = result.getDecision();

                if (effect == Result.DECISION_PERMIT) {
                  atLeastOnePermit = true;
                  permitObligations.add(result.getObligations());

                }
```

```
                if (effect == Result.DECISION_DENY) {
                    atLeastOneDeny = true;
                    denyObligations.addAll(result.getObligations());
                } else if (effect == Result.DECISION_INDETERMINATE) {
                    atLeastOneError = true;


                   // keep track of the first error, regardless of cause
                    if (firstIndeterminateStatus == null) {
                        firstIndeterminateStatus = result.getStatus();
                    }
                }
            }
        }


        if( atLeastOnePermit){
            if (!oblgCombAlgIds.isEmpty()){
                  CompoundResult cResult = new
CompoundResult(Result.DECISION_PERMIT,

                  permitObligations,policySetObligations,null);

                  Iterator itr = oblgCombAlgIds.iterator();
                  while(itr.hasNext()){

                        URI id = (URI)itr.next();
                        try{
                              OblgCombAlgFactory factory =
                                                  OblgCombAlgFactory.getInstance();
                              oblgCombAlg = factory.createAlgorithm(id);
                        }
                        catch(Exception e){
                              return new Result(Result.DECISION_INDETERMINATE);
                        }
                        cResult = oblgCombAlg.combine(cResult);

                  }
                  // collecte all the obligations that need to be returned
                  Set finalObligations = new HashSet();
                  finalObligations.addAll(cResult.getObligationSet());

                  Iterator itr1 = cResult.getObligations().iterator();
                  while(itr1.hasNext()){
                        Set oblg = (Set)itr1.next();
                        finalObligations.addAll(oblg);
                  }
                  return new Result(cResult.getEffect(),
                             context.getResourceId().encode(),
                             finalObligations);
            }
            Set originalObligations = new HashSet();
            Iterator itr2 = permitObligations.iterator();
                  while(itr2.hasNext()){
                        Set oblg = (Set)itr2.next();
                        originalObligations.addAll(oblg);
                  }
            return new Result(Result.DECISION_PERMIT,
                    context.getResourceId().encode(),
                    originalObligations);
```

```
        }

        // if we got a DENY, return it
        if (atLeastOneDeny) {
            return new Result(Result.DECISION_DENY,
                              context.getResourceId().encode(),
                              denyObligations);
        }

        // if we got an INDETERMINATE, return it
        if (atLeastOneError) {
            return new Result(Result.DECISION_INDETERMINATE,
                              firstIndeterminateStatus,
                              context.getResourceId().encode());
        }

        // if we got here, then nothing applied to us
        return new Result(Result.DECISION_NOT_APPLICABLE,
                          context.getResourceId().encode());
    }

}
```