Risk-Aware Access Control and XACML

Luca Gasparini

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Laurea Magistrale
of the
Università degli studi di Padova.

Dipartimento di Ingegneria Informatica

April 2013

Declaration

No portion of the work contained in this document has been submitted in support of an application for a degree or qualification of this or any other university or other institution of learning. All verbatim extracts have been distinguished by quotation marks, and all sources of information have been specifically acknowledged.

Signed:

Date: April 14, 2013

Abstract

Risk-Aware Access Control (RAAC) is a novel access control model that recently has emerged as an important research subject as it addresses the increasing need of securely sharing information in a dynamical environment. Given an access request, the goal of a RAAC system is to take a decision performing a dynamic risk-analysis in order to manage the trade-off between the risk incurred by granting a permission to a subject and the one incurred by not granting it. When a risky access is allowed, the system can apply some mitigation methods. These are actions that might be fulfilled by the system itself or by the requester with the goal of account for or reduce the risk incurred by granting the access. We refer to these actions as system obligations and user obligations respectively.

In this thesis, we propose an extension of an existing RAAC abstract model that supports risk assessment, risk-aware authorisation decision making and the use of system and user obligations as risk mitigation methods. We also propose an implementation of the extended abstract model based on XACML, a standard that defines an XML-based language for the specification of access control policies, requests and responses together with a reference architecture for its implementation. Specifically we define a set of XACML policies to encode the features of the RAAC model and provide a concrete implementation of the reference architecture that supports dynamical risk-assessment, and enforcement and monitoring of obligations.

We develop a novel Risk-Aware Group Based Access Control (RA-GBAC) model based on the our previous RAAC model, in order to better support the sharing of information and permissions in a dynamic collaborative environment. We introduce the notions of group and task, and provide support for risk-aware activation of tasks. Specifically, a group of users can request a set of permissions needed for the fulfilment of a common goal. A successful task activation could result in a set of obligations imposed on all the members of the requesting group, which we called collective obligations. We thus introduce support for specification and monitoring of this kind of obligation together with a mechanism to incentive group to the fulfilment of their obligations. We also propose an improved XACML RAAC architecture that supports the implementation of access control system based on our RA-GBAC model.

Acknowledgements

Firstly I would like to express my deep gratitude to my co-supervisor, Tim Norman, for giving me the opportunity to carry out my thesis project under his supervision at the University of Aberdeen, for his patience, his valuable advices and for constantly helping me improving my research and writing skills. I would like to thank my supervisor, Carlo Ferrari, for the assistance he provided during the writing process of the thesis. Special thanks goes to Liang Chen, for being such a great advisor. He constantly motivated and supported me during the project development, and helped me in all of its phases. He reviewed all my thesis drafts and gave me lots of good advices on how to improve them. I would like to thank all the officemate for making the dot.rural 912 office such a lively and friendly place.

Thanks to all the friends I met during the last year in Aberdeen; in particular Ceren and all the "Karaoke girls", Alessandro, Antonio, Andrea, the flatmates of Esslemont 112, Youlia, especially for hosting me while I was searching for accommodation, Lech and Joselia with all the "winter beach BBQ" gang, Paul, Clement and of course Francesca; you all helped me making this year in Aberdeen an awesome experience.

I'm grateful to all my friends Alessandro, Anna, Giacomo, Michele, Riccardo, Roberta, Stefano because, even if I've been away for a long time and i missed you a lot, you always made me feel like you were there with me.

Last, but not least, I would like to thank my family; I am deeply and forever indebted to my parents and my brother for their love, encouragement and financial support throughout all my studies.

Luca Gasparini
Istrana, 2013

Contents

1 Introduction 8
1.1 Motivation and overview . . . 9
1.2 Contribution . . . 11
1.3 Thesis outline . . . 11

2 Background 13
2.1 RAAC . . . 13
2.1.1 An abstract model for RAAC . . . 14
2.2 XACML . . . 15
2.2.1 Reference architecture . . . 16
2.2.2 Policy language . . . 17

3 The XACML RAAC Implementation 23
3.1 RBAC and its XACML profile . . . 24
3.2 Defining the risk mitigation strategy . . . 25
3.3 Risk assessment . . . 27
3.4 A Java implementation of our RA-XACML model . . . 28
3.4.1 Balana XACML 3.0 implementation . . . 28
3.4.2 Implementing the risk assessment module . . . 29
3.5 Extending the model and its XACML implementation . . . 31
3.5.1 Supporting user obligations . . . 31
3.5.2 Formal model . . . 32
3.5.3 XACML implementation . . . 33
3.5.4 Implementing the obligations service . . . 36
3.6 Discussion . . . 36

4 Risk-Aware GBAC 38
4.1 Running example . . . 38
4.2 Motivations and requirements . . . 39
4.3 Formal model . . . 40
4.3.1 Integrating tasks into RAAC . . . 40
4.3.2 Constructing a reference monitor . . . 41
4.3.3 Enforcing authorisation decisions . . . 43
4.4 The XACML implementation . . . 44

4.4.1 A complex risk mitigation <PolicySet> 44
4.4.2 The assigned-task <PolicySet> 47
4.5 Representing and enforcing collective obligations 47
4.6 Extended architecture 49
4.7 Discussion 49
4.8 Summary 51

5 Healthcare Scenario and Discussion 52
5.1 Healthcare example scenario 52
5.1.1 Task definition and its RMS 54
5.1.2 Team risk calculation 55
5.1.3 Budget sharing mechanisms 57
5.1.4 Security thresholds and constraints 59
5.1.5 Summary 60

6 Conclusion 61

List of Figures

2.1 XACML reference architecture 18
2.2 XACML policy language model diagram 20
2.3 Obligation attribute definition in XACML 3.0 20
2.4 Example of attribute retrieval in XACML 3.0 21
2.5 Example of request context in XACML 3.0 22
2.6 Example of response context in XACML 3.0 22

3.1 Role <PolicySet> and Permission <PolicySet> in RB-XACML 25
3.2 Role Assignment <PolicySet> 26
3.3 Risk Mitigation <Policy> 27
3.4 AttributeFinderModule 30
3.5 PolicyFinderModule 30
3.6 RiskCalculator Interface 30
3.7 The XACML architecture with risk assessment module 31
3.8 Risk mitigation strategy for permission to read Alice's SCR 34
3.9 Complex Risk Mitigation <Policy> 35
3.10 The extended architecture for supporting obligations 37

4.1 Scenario role hierarchy 38
4.2 Risk mitigation policy for task assignment 46
4.3 Assigned task policy 48
4.4 Collective deduct-budget <ObligationExpression> 48

5.1 Scenario role hierarchy 52

Chapter 1

Introduction

With the term *information security* or *computer security* we refer to a set of mechanisms which have the goal of protecting information against unauthorized accesses, uses or modifications. In the last decades, computer system rapidly became a core part of most of the organizations' information systems. Nowadays they are often used to store and process large amount of sensitive data which need to be preserved and protected from malicious access but, at the same time, to be available when needed by authorized users. Moreover, with the increasing diffusion of the Internet, that allows and promotes the dynamic exchange of information, and the proliferation of technologies like e-banking and e-commerce, *computer security* is becoming a key issue and has been receiving a lot of interest by the research community.

Access control, in security, is a mechanism that allows a system administrator to define who can perform some action or access some resources in a system and in what circumstances. The main purpose of access control is to promote optimal and secure sharing and exchange of information and resources according to the *basic principles on information protection* outlined by Saltzer and Schroeder (1975). Of those principles, the ones related to access contol are:

- *Fail-safe defaults* principle: authorization decisions have to be based on permission rather than exclusion. The default situation is the lack of authorization and the protection scheme identifies conditions under which access is permitted.

- *Least privilege* principle: at any time, a user has to be given all and only the permissions that are necessary to perform his or her job function.

- *Complete mediation* principle: every access to every object must be checked for authority.

- *Separation of privilege* principle: a protection mechanism that requires two keys to unlock it is more robust and flexible than one that allows access to the presenter of only a single key.

Three commonly known requirements that a well designed computer security system has to guarantee are:

1. *Confidentiality:* only authorized users may access some sensitive information.

2. *Integrity:* information should be protected from being improperly altered or deleted by unauthorized users.

3. *Availability:* information must be available to authorized users when needed.

Access control is critical to the satisfaction of both requirements 1 and 2. In this thesis we discuss and propose some novel access control models.

Traditional access control systems are based on policies that statically define by whom and in what circumstances a resource can be accessed. The Trusted Computer System Evaluation Criteria (TCSEC) (Department of Defense, 1985) defines two different categories of access control system: Mandatory Access Control (MAC) and Discretionary Access Control (DAC). In a DAC system, also called identity-based access control system (Bishop, 2003), every resource has an owner or initiator who can determine who may or may not access to the resource (Department of Defense, 1985). Several research work has been done on the field of DAC resulting in the definition of different models (Lampson, 1971; Graham and Denning, 1972; Sandhu, 1992; Harrison et al., 1976). In a MAC system, the policies that define who can access to every resource is defined or changed only by an higher authority (usually a system administrator). These policies are usually based on some inequality between certain attributes of the user and attributes of the resource(Sandhu, 1993). A well-know model that embodies the features of MAC is the Bell-LaPadula model (Bell and LaPadula, 1975).

In the last decade a novel access control paradigm, called Role Based Access Control (RBAC) has attracted considerable research interest. The basic idea of RBAC is to introduce the concept of a *role*, which acts as a “bridge” between users and permissions. RBAC greatly simplifies the management of permissions in an organization. More specifically, system administrators can create roles according to the job functions performed in an organization, and then, assign users to roles on the basis of their specific job responsibilities and qualifications. A set of permissions is assigned to every roles based on the work-related activities. Over the years, there has been a considerable number of models that have been developed for RBAC. The RBAC96 family of models is undoubtedly the most well known model for RBAC (Sandhu et al., 1996), and provides the basis for the recent ANSI RBAC standard (American National Standards Institute, 2004). It consists on of four conceptual models that form a hierarchy: $RBAC_0$ defines the basic features of RBAC; $RBAC_1$ and $RBAC_2$ extend $RBAC_0$ through the addition of role hierarchy and constraints respectively; and $RBAC_3$ includes all the features of $RBAC_1$ and $RBAC_2$.

1.1 Motivation and overview

We believe that the traditional access control models don’t correctly address the increasing need of a flexible access control mechanism. Specifically:

- Most often, the traditional authorisation policies are too rigid to handle the exceptional or emergence situation in which access rights should be appropriately overridden in order to proceed business functionality.
- They don’t address the requirements of dynamic secure information and permission sharing addressed by today’s collaborative environments. For example it could be reasonable, in some cases, to let a user access to some permissions to which he would normally not be allowed, when he is acting in behalf of a group of users that are collaborating for the fulfilment of a common goal.

- They are not flexible enough to deal with the changing behaviour of the users and the so called “insider threats”. If a user that is normally allowed to access to some sensitive information starts to behave in a suspicious way, the system should be able to detect these changes in his behaviour and to limit the risk incurred by denying some of the permissions associated with the user.

Recently, the design of risk-aware access control (RAAC) systems (Chen and Crampton, 2011; Chen et al., 2012; Cheng et al., 2007; JASON Program Office, 2004; Martinelli and Morisset, 2012) has become an important research field as they were proposed to address the need to enable access to resources and information in a dynamic environment. The core goal of developing RAAC is to provide a mechanism that can manage the trade-off between the risk of allowing unauthorized access with the cost of denying access when the inability to access resources may have profound consequences. Security systems based on RAAC are considered to be more permissive as they allow some risky access that would be denied by traditional access control systems.

We can distinguish two different phases in the request evaluation process of a RAAC system:

1. The risk associated with the requested is assessed by the system. This phase usually returns a numeric value that represents the risk incurred by the system allowing the request. When assessing the risk value, many different factors should be considered by the system, like, for example, the trustworthiness of the requesting user, the sensitivity of the data that was requested and the criticality of the action that the user wants to execute.

2. Given the result of the risk assessment, the system has to decide whether to grant or to deny the permission. The system administrator usually can define, for each permission, a maximum risk threshold. If the assessed risk-value lies below the defined threshold the system grants the permission and eventually takes additional actions.

Usually, when a risky access is allowed, the system apply some *risk mitigation methods*. These methods could include actions to be performed by the system itself to which we refer as *system obligation*, or other actions referred to as *user obligations* that the user must perform before or after the access request has been granted. An example risk mitigation method performed by the systems could be logging requests for future review. User obligations can include requiring the user to write a report in which he justify the requests, or to delete certain sensible data after use.

In January 2013 the OASIS standard organization proposed the XACML 3 (OASIS, 2013) as a standardized access control policy language. Since the release of XACML 2 (OASIS, 2005b), the standard has attracted lots of interest from the research community in the field of the development of traditional access control system as it promotes the separation of the decision point from the enforcement point, allowing to re-use the same policies in different applications. Specifically, the standard defines an XML-based language for the specification of access control policies, requests and responses, together with a reference architecture for the implementation of an XACML-based access control system. It supports the definition of Attribute Based Access Control (ABAC) policies and it provides policy profiles for the definition of different traditional access control models (e.g. RBAC XACML profile (OASIS, 2010)). However XACML doesn’t naturally support risk-assessment and we believe that the actual reference architecture doesn’t allow to easily implement RAAC system using the only standard feature of the policy language.

1.2 Contribution

The main contributions of this thesis can be summarized as follows:

- Firstly we propose an implementation of an XACML-based RAAC system based on the work by Chen et al. (2012), extending the reference XACML architecture in a modular way. Unlike Chen et al. (2012), in this first implementation we only support the use of system obligation as risk mitigation methods. Specifically we implement an external risk-assessment module that dynamically assesses the risk of a request and allow us to refer to this value as a normal attribute. We show how to write standard XACML policies that implements the risk mitigation strategies defined in the RAAC abstract model.

- We propose an extended abstract model for risk-aware access control systems, based on the work by Chen et al. (2012). This model supports both system and user obligations as mitigation methods. Specifically, for every risk range of a risk mitigation strategy, we can specify, a set of system and user obligations to be returned by the decision point. Since user obligations, unlike system obligations, are not guaranteed to be fulfilled, we need to introduce a mechanism that encourages the subjects in fulfilling their obligations. For this reason, we introduce, for each subject a budget value. When he gets access to a permission for which he needs to fulfil some obligations, a subject will have to deposit some of his budget. In order to get back the deposit he will have to fulfil all of the returned obligations before a specified deadline.

- We extend our XACML-based RAAC implementation to support the extended model. We introduce support for the deposit of budget value and the specification of user obligations in the risk mitigation strategies. We implement a user obligation fulfilment monitor in order to decide whether to give back or not the budget value to the requester.

- We then develop risk-aware group-based access control (RA-GBAC) based on previous models in order to better support the management of permissions in collaborative environments in which groups of users collaborate to fulfil a common task. We introduce the *task* as an entity that groups together all the permissions necessary for the whole time-period needed to fulfil it. When a team gets the permission for fulfilling a task, all the members can access to all the permission associated with the task for the whole time period. We adapt the concept of mitigation strategy and risk assessment to be applied for the assignment of a task to a group of users. We introduce the collective obligations as risk mitigation method that have to be fulfilled by one of the member of the requesting team. Team members can share their budget and use part of it for activating some tasks.

- We extend our XACML-aware architecture to encode the group model by introducing the task and group entities. We adapt the definition of the risk-assessment procedure for the assignment of a task to a team. We introduce a budget-sharing mechanism and the monitoring of collective obligations.

1.3 Thesis outline

The remainder of this thesis is organized as follows:

- Chapter 2: We present some background knowledges about RAAC and XACML. Specifically, we introduce RAAC, and then we present in a formal way an abstract RAAC model based on the one defined by Chen et al. (2012). In this model we restrict the set of supported risk mitigation methods only to system obligations. We do so in order to firstly provide a simple implementation of this model that can be extended later to support additional features. We then introduce the syntax and the main features of XACML policy language and the reference architecture for the implementation of an XACML-based system.

- Chapter 3: We firstly provide an XACML implementation for the RAAC model presented in chapter 2 showing how it can be used on top of the RB-XACML profile. We extend the XACML reference architecture and show a policy example in which we represent the risk mitigation strategy for the activation of a permission. We then extend the RAAC model instantiating the abstract model defined by Chen et al. (2012). We introduce the support for user obligations and the concept of budget and show how to extend the XACML implementation to support these new features.

- Chapter 4: We propose a risk-aware group-based access control (RA-GBAC) model. We extend the XACML implementation of Chapter 3 and show an example of policy that defines a risk mitigation strategy for the activation of a task by a group. We discuss the implementation of a modular obligation monitor that support collective obligations.

- Chapter 5: We present an example based on an healthcare scenario to underline the motivation for the introduction of our RA-GBAC model. While discussing the example we make some remarks on the possible coherent choices for a risk-mitigation strategy associated with a task, the risk assessment algorithm and the budget sharing mechanism. We also discuss the possibility to define some additional security constraints and risk thresholds.

Chapter 2

Background

In this chapter we give an overview of the background knowledge related to the focus of this thesis. First, we introduce the risk-aware access control (RAAC) paradigm, how it works and the motivations for its use. We then present the abstract model for RAAC proposed by Chen et al. (2012) which underpins our work in the following chapters. After that we introduce the eXtensible Access Control Markup Language (XACML) standard that will be used later to implement our RAAC system.

2.1 RAAC

RAAC is a novel access control paradigm designed to enable optimal access to resources and sharing of information in a dynamic or collaborative environment. The requirements for what we refer as optimal access to resources are addressed by the *principle of least privileges* (Saltzer and Schroeder, 1975). This principle states that a subject, at any time, must be allowed to access only the information and resources that are necessary to fulfil his task. Access control systems are usually implemented by an *authorization decision function* which, given an access request from a user, returns a decision. Authorization decision functions are usually based on the access control list mechanism in which every permission is associated with a list of access control entries, each of which identifies a subject and a set of access rights. This approach requires a prior definition of policies that specifies a set of authorised requests, but does not meet the requirements of dynamic environments where the access needs are often unpredictable.

In a RAAC system the decisions are taken according to the result of a risk analysis by the system for the given access request. In fact, the core goal of developing RAAC systems is to provide a mechanism that can manage the trade-off between the risk of allowing unauthorized access with the cost of denying access when the inability to access resources may have profound consequences. This approach can be particularly useful, for example, to allow some risky access in an emergency situation, when the risk incurred by allowing some access is considered acceptable compared to the consequences of denying the access. Security systems based on RAAC are generally considered to be more permissive as they allow some risky access that would be denied by traditional access control systems.

The principal issues in developing a RAAC system are:

1. How to assess the risk of granting an access request; and
2. How to decide, given the risk value, whether to allow the risky access or to deny it.

When a risky access is allowed, the system could require a user, or the system itself to perform some action with the goal of accounting for and reducing the risk incurred by granting the permission. We refer at these action as *risk mitigation methods*. In the case that these actions can be performed by the system, we refer to them as *system obligations*. On the other hand, if these actions have to be performed by the user we refer to them as *user obligations*. An example system obligation could be logging requests for future review. User obligations can include requiring the user to write a report in which he justifies the requests, or to delete certain sensitive data after use. While the access control system can guarantee that the system obligations will be performed, a user could intentionally or unintentionally fail to fulfil his obligation. For this reason access control systems often introduce some mechanism to encourage the users to fulfil their obligations. In traditional access control system the returned decision usually can assume only the values `permit` or `deny`. Since RAAC systems, with the introduction of risk mitigation methods, can return richer type of decision response, they are considered more flexible than the traditional systems.

Chen et al. (2012) proposed an abstract model for RAAC defining basic components of a RAAC system that supports risk assessment, risk-aware authorisation decision making and the use of system and user obligations as risk mitigation methods. In the next section we are going to formally introduce a RAAC module based on their work that supports only system obligation as risk mitigation methods and that will be extended in the next chapters to support some additional features.

2.1.1 An abstract model for RAAC

Let P be a set of permissions. A permission represents an action-object pair for which a subject may be authorised. Let S be a set of subjects. A subject represents an active entity in a system that may request access to resources. Let C be a set of contexts. We model an access request as a tuple $\langle s,p,c\rangle$, where $s \in S$, $p \in P$ and $c \in C$. In general, the risk of granting a permission to a subject in a particular context can be interpreted as the likelihood of the permission being misused by the subject. Determining the likelihood of misuse depends on various factors such as the security attributes of the subject (e.g. trustworthiness, roles or access history), the value of the resource, the context (e.g. device, location or current time) from which the subject is requesting access, etc. Let Σ denote a set of states, and $\mathcal{K} = \{k \in \mathbb{R} : 0 \leqslant k \leqslant 1\}$ denote a risk domain. We define a risk function $\mathsf{Risk} : Q \times \Sigma \to \mathcal{K}$ that takes as input an access request $q = \langle s,p,c\rangle \in Q$ and the current system state $\sigma \in \Sigma$, and returns the risk $k \in \mathcal{K}$ associated with the request. There are a number of ways of explicitly defining the Risk function depending on system requirements and a concrete access control model. These are domain-dependant, and thus outside the scope of this paper.

From the system's perspective, we need to determine a risk threshold that the system is willing to accept when granting access requests, and what kind of risk mitigation should be put in place if risky access is allowed. We define risk thresholds and risk mitigation strategies on a per-permission basis. We write $[k,k')$ to denote the *risk interval* $\{x \in \mathcal{K} : k \leqslant x < k'\}$. Let $\mathcal{O}^s$ denote a set of *system obligations*, where $o^s \in \mathcal{O}^s$ is some action that must be taken by the system when enforcing an access control decision. Then we define a *risk mitigation strategy* to be a list $[\langle k_0, O^s_0\rangle, \langle k_1, O^s_1\rangle, \dots, \langle k_{n-1}, O^s_{n-1}\rangle, \langle k_n, O^s_n\rangle]$, where $0 = k_0 < k_1 < \cdots < k_n \leqslant 1$ and $O^s_i \subseteq \mathcal{O}^s$. Let M denote a set of risk mitigation strategies. We define a function $\mu : P \to M$, where $\mu(p)$ denotes

the risk mitigation strategy associated with permission p. Informally, a risk mitigation strategy $\mu(p)$ for $p \in P$ specifies that obligations O_i^s will be executed if the risk of granting p is within the interval $[k_i, k_{i+1})$. Note that a special case of our approach is to define a single risk mitigation strategy that is applicable to all permissions, and this is the approach advocated in Cheng et al. (2007) work.

Formally, given a request $q = \langle s, p, c \rangle$ and a system state σ, we define an authorisation function Auth as,

$$\mathsf{Auth}(q, \sigma) = \begin{cases} \langle \mathsf{allow}, O_i^s \rangle & \text{if } \mathsf{Risk}(q, \sigma) \in [k_i, k_{i+1}),\ 1 \leqslant i < n, \\ \langle \mathsf{deny}, O_n^s \rangle & \text{otherwise.} \end{cases}$$

In other words, the request $\langle s, p, c \rangle$ is permitted but the system must enforce obligations O_i^s if the risk of allowing $\langle s, p, c \rangle$ belongs to $[k_i, k_{i+1}]$, and the request $\langle s, p, c \rangle$ is denied but the system must perform O_n^s if the risk is greater than or equal to k_n.

We believe that these risk-based features can be naturally integrated into existing access control models, making them become risk-aware. In role-based access control, for example, we may introduce risk assessment on user-role activation. In this case, a subject $s \in S$ is regarded as a user or a session, and a permission $p \in P$ as an approval to activate a particular role. Of course, there exist other possible interpretations of subjects and permissions for RBAC or other access control models. In most cases, a permission is thought of as an approval to perform an operation on a protected resource, and this is the notion defined in the RBAC standard (American National Standards Institute, 2004), whereas a subject could also be regarded as a role or even a security group.

In order to illustrate the features of RAAC, we introduce a concrete example for accessing patient records in an emergency situation. One evening, Alice is knocked unconscious in a car accident and is taken into the emergency department by an ambulance. The emergency doctor treating her, Bob, would like to view her summary care record (SCR) in order to find out whether there are any important factors to consider, such as any allergies to medications. However, Bob is not allowed to access the SCR via the current activated `Doctor` role. In this case, Bob attempts to activate `EmergencyDoctor` role, and the system determines whether to grant this request based on risk assessment. The risk computation depends on two factors associated with the request: the level of competence of Bob to activate this role, and the context (e.g. emergency situation) in which the request was submitted. Eventually, the system deems the risk is acceptable and allows Bob to activate the `EmergencyDoctor` role, thereby allowing him to access Alice's SCR. Meanwhile, all those activities are noted in an audit trail, and result in an alert being automatically sent to a privacy officer.

2.2 XACML

XACML (eXtensible Access Control Markup Language) is an OASIS standard that defines an XML-based language for specifying access control policies, requests and responses. The standard also define a reference architecture for implementing an access control system based on this language. The latest version XACML 3.0 (OASIS, 2013) was ratified by OASIS standards organization in January 2013. The XACML policy language is designed to support ABAC (Attribute

Based Access Control). Decisions can be made according to some conditions on the attributes of the subject, the object of the request or the system environment. This approach makes XACML a flexible authorization system that allows the specification of context-aware and risk-intelligent access control policies. The standard also defines a RBAC (Role Based Access Control) profile as a particular case of an ABAC system. Another important feature of XACML consists of the capability of a policy to refer to some other policies, combining their results according to different policy-combination-algorithms. This allows the implementation of distributed security systems in which resources and policies can reside in different locations or organizations. The capability of an XACML policy to refer to other policies also supports features like role hierarchy or the re-use of the same policy for controlling access to different kinds of resources, avoiding inconsistencies and eliminating duplications between policies.

XACML promotes the separation of the authorization decision point from the decision enforcement point. This separation makes it easier for a system administrator to manage access control policies for the whole system. In fact the standard architecture offers authorization as a service to the system. Every application just needs to query the XACML decision point to know whether or not it is allowed to perform an action. There is therefore, no need to embed the access control rules in the code of different applications. When a system administrator wants to change some access control rules he will just need to adapt the related policy in the XACML authorization system and this modification will affect all the applications of the information system.

Here we describe the specification for the reference architecture and the policy language defined in the version 3.0 of the XACML standard, that is the one used in our implementation.

2.2.1 Reference architecture

The reference XACML architecture proposed by OASIS is shown in the data-flow diagram in Figure 2.1. We now describe its main components referring to the typical data-flow generated by an access request in an XACML system.

- *PDP* (Policy Decision Point): receive access requests from the context handler (step 3) and it queries the Policy Repository to find the applicable policies for an access request (step 4). This is done by evaluating the applicability conditions over the request context. Only one applicable condition at a time can be found for every access request. If no, or more than one applicable policy is found the PDP has to return a response containing an error code. When it finds an applicable policy, the PDP has to evaluate its rules in the way that we will explain later and returns the corresponding decision to the context handler (step 9). The evaluation of the rules and the policy applicability conditions can require the PDP to retrieve some additional attributes not specified in the request context. These attributes can be retrieved by the context handler querying an apposite module called PIP (steps 5-8).

- *PEP* (Policy Enforcement Point): is an application-dependent component. The standard states that it has to intercept every user access request (step 1), forward it to the context handler (step 2) and enforce the authorization decision returned by the context handler (step 10). Enforcing the returned decision could involve the fulfilment of some system obligations returned in the response. In particular the specification for the reference architecture states that the returned decision can be enforced only if all the obligations can be correctly

interpreted and fulfilled by the PEP. In case this is not possible the PEP shall return an error message. As shown in the data-flow diagram (Figure 2.1) (OASIS, 2013) the system obligation fulfilment is usually performed by a separate module of the PEP.

- *Context handler*: Receive the requests from the PEP in an application-dependent format (step 2), convert them to an XACML request and forward them to the PDP adding the request context information (step 3). It can be asked by the PDP to retrieve some additional attribute from the PIP if these are needed to evaluate a particular policy or rule (steps 5-6-7-8). During this process, the context handler may have to identify the particular entity to which the attribute query refers. This information is usually included in the request context as an identifier attribute for the entity. The context handler is also responsible for receiving the XACML response from the PDP, with system obligations, and to convert it to a format understandable for the PEP before forwarding it (steps 9-10).

- *PIP* (Policy Information Point): Provides the PDP with all the information not available in the request context that is needed in order to evaluate an access request (e.g. subject, resource and environment attributes). It acts as an interface between the PDP and the Data Source of the application. Since the data source interface can differ for every application, this is also defined as an implementation-dependent component. In fact the PIP may be part of the application, such as a username/password file, or external to the application, such as a SAML (OASIS, 2005c) attribute authority.

- *Policy Repository* or *PAP* (Policy Administration Point): It stores all the XACML policies. It is queried by the PDP every time it has to find an applicable policy for an XACML access request or to find a specific policy that has been referred by another one. An important feature of a PAP is that it has to allow the definition of a subset of the policy set as top-level policies. These are the only policies that directly applicable for evaluating a request. The policies in the set of non-top-level policies can be used only if referred to by another one. It is important to carefully define these two sets in a way that, for every possible request, only one applicable top-level policy can be found in the PAP.

2.2.2 Policy language

We now describe the XACML language model for policies, request and responses. The policy specification language model is showed in Figure2.2 (OASIS, 2013). In XACML the atomic unit of an access control policy set is a `<Rule>`. A XACML `<Policy>` may be composed of a number of individual rules combined together according to a combining algorithm. In the same way a `<PolicySet>` may be composed by a number of individual policies. Three top-level policy elements are defined:

- `<Rule>`: contains a boolean condition c. If c evaluates to `true`, the rule returns a specified decision $d \in \{\text{“Deny”}, \text{“Permit”}\}$. If the condition evaluates to `false` the rule returns `Not applicable`. A rule cannot be used in isolation to determine an authorization decision, but it is intended to exist only within a `<Policy>` element. The main components of a rule are:

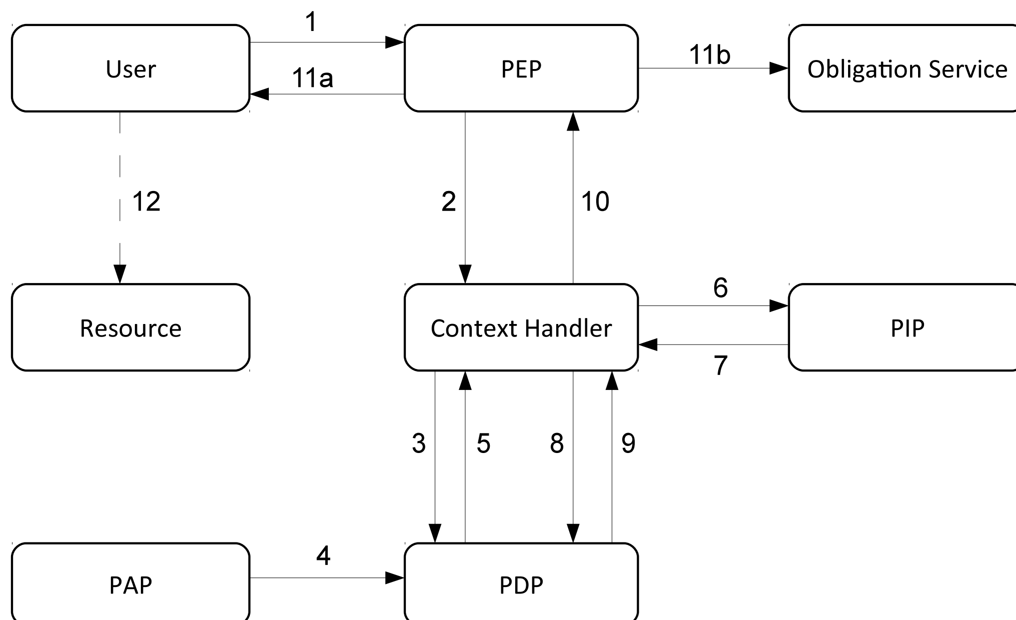

Figure 2.1: XACML reference architecture

- A `<Target>`: a simple condition in which one or more attribute values match with a predefined value. It specifies for which requests the rule applies. If it is not specified, the target is the same as the one from the `<Policy>` element to which it belongs, if this is present, otherwise it matches with every request. The target can only contain a disjunctive sequence of `<AnyOf>` elements. Every `<AnyOf>` element can only contain a conjunctive sequence of `<AllOf>` elements. Every `<AllOf>` element can only contain a disjunctive sequence of `<Match>` elements. A `<Match>` element compares an attribute value with an embedded value applying a specific match function. For the rule element to be applicable to the decision request every `<AnyOf>` element has to evaluate to true. For a `<AnyOf>` to evaluate to true, at least one of the contained `<AllOf>` elements has to evaluate to true. For a `<AllOf>` element to evaluate to true, every contained `<Match>` element has to evaluate to true.

- a `<Condition>`: a boolean expression that refines the applicability of the rules beyond the predicates implied by its target. It allows the system administrator to write boolean expressions that apply functions chosen from a standard set to one or more attribute variables. A `<Condition>` element must contain one element of a type that extends the `<Expression>` type. The following elements are subtypes of the `<Expression>` type: `<Apply>` and `<Function>` that apply a specified function to a number of members; `<AttributeDesignator>` and `<AttributeSelector>` that allows the retrieval of the value of a named attribute from the request context and from a given xml path respectively; the `<VariableReference>` element that allows reference to a variable defined using the `<VariableDefinition>` which we will describe later.

- An `Effect` attribute: the decision value $d \in \{\text{“Deny”}, \text{“Permit”}\}$ that has to be returned if both the rule and target conditions evaluates to true. Note that, if the condition or the target doesn't apply the rule evaluation will return “Not applicable”.
- A set of `<ObligationExpression>` elements: that specify some system obligation that has to be fulfilled by the PEP when the rule applies and the specified effect is returned. All the elements of this set have to be enclosed between the `<ObligationExpressions>` element. An obligation in XACML is represented as a set of attribute assignments. No semantics are defined for the obligation specification, the standard just states that the PEP has to be able to interpret the specified obligation. In Figure 2.3 we show an example system obligation that obliges the PEP to send a confirmation email to the user who obtained access to a resource. The `<ObligationExpression>` element must contain a set of `<AttributeAssignmentExpression>` each one assigns a value to a specified `AttributeId`. The value is enclosed in the tag and has to be a subtype of the `<Expression>` element type previously defined.
- A set of `<AdviceExpression>` elements: that specify some system advices that the PEP should fulfil when the rule applies and the specified effect is returned. They have been introduced since the last version of the standard and they follow the syntax defined for the `<ObligationExpression>` element.

- `<Policy>`: contains a set of `<Rule>` elements and a specified procedure for combining the results of their evaluation. It is the basic unit used in the policy specification and so it is intended to form the basis of an authorization decision. The main components of a `<Policy>` are:

  - A `<Target>`: defined in the same way as the Rule's `<Target>`.
  - A set of `<VariableDefinition>` elements: these allow the policy writer to associate a `VariableId` to a specific value. The value is enclosed in the tag and has to be a subtype of the `<Expression>` element type previously defined. These variables can be referenced using the `<VariableReference>` throughout the whole policy in the definition of rule's condition, obligations or advice.
  - A set of `<Rule>` elements.
  - A `rule-combining-algorithm` attribute: it specifies the algorithm used by the PDP to combine the results of the different rule components. For example, in the case that the specified combining algorithm is `deny-override`, and if at least one `<Rule>` element which evaluates to deny is encountered, the returned decision will be `Deny`, no matter what are the returned decisions of the other rules are. XACML identifies six combining algorithms, but also provides an extension point that allows us to define new algorithms.
  - A set of `<ObligationExpression>` elements: as specified for the `<Rule>` element.
  - A set of `<AdviceExpression>` elements: as specified for the `<Rule>` element.

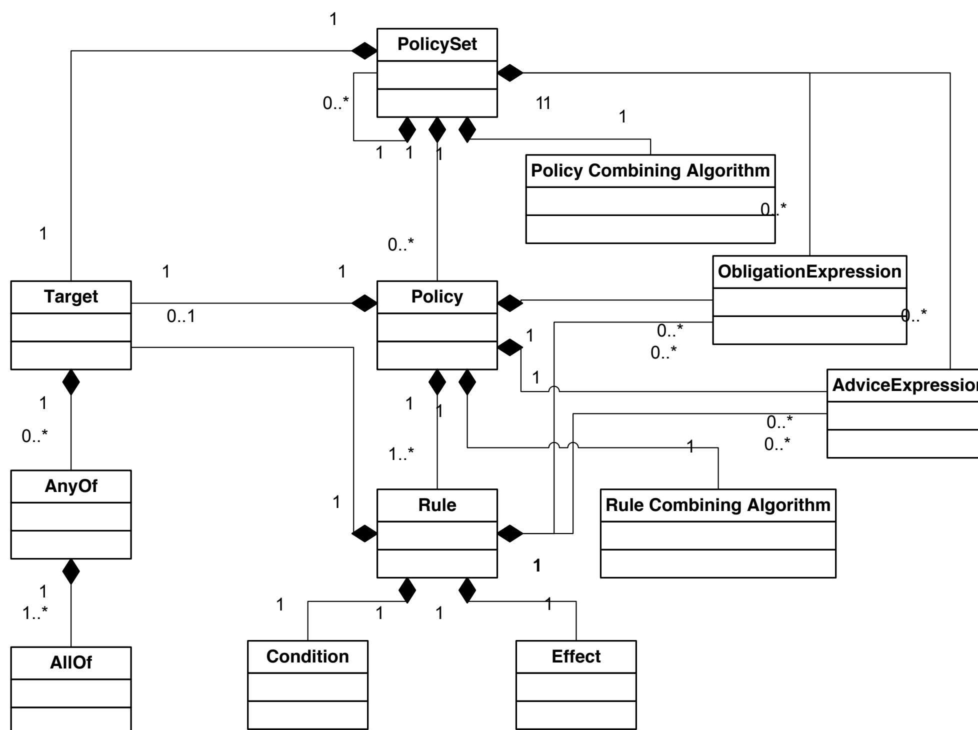

Figure 2.2: XACML policy language model diagram

- <PolicySet>: contains a set of <Policy> or other <PolicySet> elements and a specified procedure for combining the results of their evaluation. This is the standard means for combining separate policies into a single combined policy. The main components of a PolicySet are:
  - A <Target>: as specified above.
  - a set of <Policy> or <PolicySet> elements. An external <PolicySet> or <Policy> can be referred by its id using respectively the <PolicySetIdReference> and the <PolicyIdReference>
  - A Policy-combining algorithm: specifying the algorithm used by the PDP to combine the results of the different Policy components. These algorithms are defined in the same way as for the rule-combining-algorithms.
  - A set of <ObligationExpression> elements: as specified above.
  - A set of <AdviceExpression> elements: as specified above.

```
00   <ObligationExpression ObligationId="send-confirm-email" FulfillOn="Permit">
01      <AttributeAssignmentExpression AttributeId="mail"
02         DataType="http://www.w3.org/2001/XMLSchema#string">
03         <AttributeDesignator AttributeId="subject-category:mail"
04            Category="urn:oasis:names:tc:xacml:1.0:subject-category:access-subject"
05            DataType="http://www.w3.org/2001/XMLSchema#string" MustBePresent="false" />
06       </AttributeAssignmentExpression>
07   </ObligationExpression>
```

Figure 2.3: Obligation attribute definition in XACML 3.0

As we stated before, this hierarchical structure allows XACML to support distributed policy systems, with some policies that could be stored and evaluated in different servers that could act as distinct PDPs. The PDP server that receives an access request for some distribute resources will have to ask the different servers to evaluate the request and then combine the decisions applying the specified combining algorithm.

We mentioned before that one of the subtypes of the `<Expression>` element type is the `<AttributeDesignator>` element. This tag is used to refer to an attribute value. An attribute is identified both by an `AttributeId` and a `Category`. XACML defines three different categories for variables of subjects, resources and actions of the access request and one for environment variables (e.g. date, time). The context handler is responsible for returning to the PDP the value associated with the variable, searching for it in the request context or querying the PIP. From version 3.0 of XACML it is possible to define custom categories of attributes. These new categories have to be supported by the application PIP. In Figure2.4 we show how to use the `<AttributeDesignator>` element to retrieve the id value of the subject of the request. In Line 01 and 02 we specify the id and the category of the attribute respectively. The attribute has to be of one of the types supported by XACML, and this type has to be specified in the `DataType` (Line 03).

```
00   <AttributeDesignator
01      AttributeId = "urn:oasis:names:tc:xacml:1.0:subject:subject-id"
02      Category= "uurn:oasis:names:tc:xacml:3.0:attribute-category:subject"
03      DataType= "http://www.w3.org/2001/XMLSchema#string"
04   />
```

Figure 2.4: Example of attribute retrieval in XACML 3.0

An XACML *request context* is created by the context handler when it receives a request from the PEP. It includes one or more sets of attribute elements, each of which is associated with one of the supported attribute categories. In Figure2.5 we show an example of a request context. All the attributes in the same category are enclosed in the same `<Attributes>` element and the category id is specified with the `Category` element. An attribute assignment is defined by the `<Attribute>` element and with the id specified as an attribute and the value enclosed in an `<AttributeValue>` element.

An XACML *response context* includes one or more results, each of which is comprised of a decision and, optionally, obligations and advice. The decision may be to permit or deny access, or to indicate that no policies or rules are applicable to the access request, or that some error has occurred. In Figure 2.6 we show an example of response context. The decision `Deny` is specified in Line 02. The `<StatusCode>` in Line 04 allows the specification of whether an error occurred. A free form message describing the status can be specified in the `<StatusMessage>` element. From Line 08 to 15 we can see an example of a returned obligation that obliges the system to audit the denied access request specifying as obligation attributes the subject and the resource of the request.

```
00 <Request ...>
01    <Attributes Category="...:subject-category:access-subject">
02       <Attribute IncludeInResult="false"
03         AttributeId="...:subject:subject-id">
04          <AttributeValue
05            DataType="urn:oasis:names:tc:xacml:1.0:data-type:rfc822Name">
06              bs@simpsons.com</AttributeValue>
07       </Attribute>
08    </Attributes>
09    <Attributes Category="...:attribute-category:resource">
10       <Attribute IncludeInResult="false"
11         AttributeId="urn:oasis:names:tc:xacml:1.0:resource:resource-id">
12          <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#anyURI">
13            file://example/med/record/patient/BartSimpson</AttributeValue>
14       </Attribute>
15    </Attributes>
16    <Attributes Category="...:attribute-category:action">
17       <Attribute IncludeInResult="false"
18         AttributeId="urn:oasis:names:tc:xacml:1.0:action:action-id">
19          <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">
20             read</AttributeValue>
21       </Attribute>
22    </Attributes>
23 </Request>
```

Figure 2.5: Example of request context in XACML 3.0

```
00 <Response ...>
01    <Result>
02       <Decision>Deny</Decision>
03       <Status>
04          <StatusCode Value="urn:oasis:names:tc:xacml:1.0:status:ok"/>
05          <StatusMessage>permission denied</StatusMessage>
06       </Status>
07       <Obligations>
08          <Obligation FulfillOn="Permit" ObligationId="audit-request">
09             <AttributeAssignment DataType="http://www.w3.org/2001/XMLSchema#string"
10               AttributeId="subject">Admin
11             </AttributeAssignment>
12             <AttributeAssignment DataType="http://www.w3.org/2001/XMLSchema#string"
13               AttributeId="resource">readme.txt
14             </AttributeAssignment>
15          </Obligation>
16       </Obligations>
17    </Result>
18 </Response>
```

Figure 2.6: Example of response context in XACML 3.0

Chapter 3

The XACML RAAC Implementation

As we described in Section 2.1, RAAC is a novel paradigm that allows an access control system to take a decision on an access request according to the result of a risk analysis that considers the various elements of the access request (user, resource) and the state of the system itself. This approach leads to a more flexible access control system able to deal with the increasing needs to exchange information and enable the access to permission in a dynamic environment.

XACML (introduced in Section 2.2) is a standard access control policy language that, in the last years, has attracted a lot of interest from the research community and some IT vendors because of its flexibility and of its architecture, compliant with the standard authorization service architecture (ISO 10181, 1996) that promotes the separation of the authorization decision point from the decision enforcement point. Our goal is to provide an XACML implementation of a RAAC model which illustrates a flexible way to make many existing system become risk aware only with minor changes in the architecture of the XACML authorization system. This is due to the hierarchical structure of the XACML policy language that allows reference from a top-level policy to other policies and use their returned decisions as partial results for its evaluation.

The XACML standard specification defines some profiles for implementing common features of access control systems, such as a role based access control (RBAC) profile, hierarchical resource profile, Security Assertion Markup Language (SAML) profile, etc. There are no standard profiles for risk-aware access control implementations yet and we are not aware of any other open-source XACML policy model that support risk-aware features. For these reasons we propose our XACML implementation of an RAAC system based on the abstract model previously defined. As stated before, our purpose is to define a generic XACML model that can be used in combination with any existing implementation to make it become risk-aware. Additionally, RBAC is one of the most widely used access control models in practice. For ease of illustration, in this chapter, we introduce our approach to implementing RAAC based on the standard RBAC XACML profile. In particular, our approach proposes the risk mitigation strategy to be associated with user-role activation, which embodies a natural way to make the XACML RBAC profile become risk-aware.

One of the main issues in developing a XACML-based RAAC model is due to the fact that the risk assessment is a dynamic procedure. The risk is in fact not a static attribute of an entity, but it can depend on the attributes of the particular request and on the status of the system when the request is received. In the following sections we will show how it is possible, using only standard features of the XACML language, and introducing only minor improvements in the implementation of some of the modules of the XACML reference architecture. Additionally, we show how

we can write risk mitigation policies that can be referenced many times in the definition of permission policies. From Section 3.4.1 we introduce an open source implementation of the XACML 3.0 architecture called "Balana" and show how it can be extended for supporting dynamic risk calculation and the definition of our risk mitigation strategies.

3.1 RBAC and its XACML profile

Role-based access control (RBAC) is an access control model that attracted a lot of interest from the research community in the last decades. The basic idea of RBAC is to introduce the concept of a *role*, which acts as a "bridge" between users and permissions. More specifically, system administrators can create roles according to the job functions performed in an organization, and then assign users to roles on the basis of their specific job responsibilities. Every role can then be assigned to a set of permissions that are necessary to perform work-related activities. This feature of a role-based model greatly simplifies the management of permissions in an organization. For example, by simply changing the set of permissions associated with a role we change the permissions set for all the users empowering that role. Moreover when a new user joins a company, we can just add him to the lists of users empowering a few roles associated to his job-functions without needing to specify every single permission that he needs. The RBAC96 family of models is undoubtedly the most well known model for RBAC (Sandhu et al., 1996), and provides the basis for the recent ANSI RBAC standard (American National Standards Institute, 2004). It introduces support for role hierarchy and additional constraints between roles, allowing the implementation of, for example, the Separation of Duty (SOD) principle.

The XACML specification defines the XACML RBAC profile (RB-XACML). The version 3.0 of RB-XACML was approved in August 2010 (OASIS, 2010). It doesn't introduce any new functionality, in fact it only updates the RB-XACML 2.0 (OASIS, 2005a) to comply with the XACML 3.0 specification. It is designed to address the core and hierarchy components of RBAC. It describes how to write a `<PolicySet>` for the definition of permission and role assignment. A Role `<PolicySet>` (RPS) associates a role identifier with a single permission `<PolicySet>` (PPS) referencing to it with `<PolicySetIdReference>` element. The target of an RPS limits the applicability of the `<PolicySet>` to subjects holding the associated role attribute and value. The PPS contains the actual permissions for a given role. It contains `<Policy>` and `<Rule>` elements that define which resources and under which conditions can be accessed by the users empowering the given role. A given PPS can also refer to other PPS from a junior role using the `<PolicySetIdReference>` element, allowing the given role to inherit all the permission of the junior role. Since a given PPS can be referenced not only by different RPS, associated with different roles, the target of the PPS doesn't have to restrict the role of the request's subject. Figure 3.1 shows pseudo Role `<PolicySet>` and Permission `<PolicySet>` that implement the emergence example described in Section 2.1.1. The `EmergencyDoctor`'s Role `<PolicySet>` (lines 01-04) consists of a `<Target>` element (line 02) and a `<PolicySetIdRef>` element (line 03). The `<Target>` only matches users who are permitted to enable the `EmergencyDoctor` role attribute. The `<PolicySetIdRef>` points to a Permission `<PolicySet>` (lines 07-14) associated with the `EmergencyDoctor` role, which implements permissions-role assignment for that

role. The Permission <PolicySet> of the EmergencyDoctor role contains a <Policy> (lines 08-12) and a <PolicySetIdRef> (line 13) elements. The <Policy> specifies a permission for reading Alice's summary care record. The <PolicySetIdRef> points to the Permission <PolicySet> associated with the normal Doctor role. This has the effect of making the permissions of the Doctor role available to the EmergencyDoctor role, thereby implementing role inheritance.

```
00 <!-- Role <PolicySet> -->
01 <PolicySet ... PolicySetId="RPS:emergencydoctor:role"...>
02   <Target>Any user with "EmergencyDoctor" attribute</Target>
03   <PolicySetIdRef>PPS:emergencydoctor:role</PolicySetIdRef>
04 </PolicySet>
05
06 <!-- Permission Set -->
07 <PolicySet ... PolicySetId="PPS:emergencydoctor:role"...>
08   <Policy ... PolicySetId="PP:scr:read"...>
09     <Rule RuleId="scr:read" Effect="Permit">
10       <Target>read Alice's SCR</Target>
11     </Rule>
12   </policy>
13   <PolicySetIdRef>PPS:doctor:role</PolicySetIdRef>
14 </PolicySet>
```

Figure 3.1: Role <PolicySet> and Permission <PolicySet> in RB-XACML

RB-XACML states that "a role attribute for a given user is a valid assignment at the time the access decision is requested, and the assignment of role attributes to users... is outside the scope of the XACML PDP" (OASIS, 2010). The standard specification suggests the use of an external *role enabling authority* (REAs) to determine the values of a user's role attributes. How the REA should work is not specified but, one possible suggestion is that it could act as a separate PDP using the set of role assignment <PolicySet> to determine whether a user can be assigned to a particular role. A role assignment <PolicySet> defines which roles can be enabled or assigned to which subjects. It may also specify restrictions on the combination or number of roles that can be enabled for a given subject.

In order to comply with RB-XACML, we believe that it is most natural to define risk assessment and risk mitigation in conjunction with Role Assignment <PolicySet> to implement risk-aware RBAC using XACML. This is also consistent with the way of defining risk mitigation strategies in the abstract RAAC model, where each resource is associated with a risk mitigation strategy, and the resource is considered as a permission to enable a role attribute for users in this case. Figure 3.2 shows a fragment of a Role Assignment <PolicySet> that comprises a <Target> element (lines 02-06) and a <PolicyIdRef> element (line 07). The <Target> determines the <PolicySet> and is only applicable to subjects who has a particular attribute (their email name is in the "nhs.com" namespace). It also restricts the resource and action attributes in the request to be EmergencyDoctor role and EnableRole respectively. The <PolicyIdRef> points to a Risk Mitigation <Policy> that further prevents subjects from enabling the EmergencyDoctor by assessing the risk of their requests.

3.2 Defining the risk mitigation strategy

We define a risk mitigation strategy in a Risk Mitigation <Policy> that is treated as a first-class entity. In other words, a Risk Mitigation <Policy> can be re-used in any Role Assignment

```
00 <!-- Role Assignment <PolicySet> -->
01 <PolicySet PolicySetId="emergencydoctor:role:requirements"...>
02   <Target>
03     <AnyOf><AllOf><Match>has email address *@nhs.com</Match></AllOf></AnyOf>
04     <AnyOf><AllOf><Match>EmergencyDoctor</Match></AllOf></AnyOf>
05     <AnyOf><AllOf><Match>EnableRole</Match></AllOf></AnyOf>
06   </Target>
07   <PolicyIdRef>rm:emergencydoctor:audit</PolicyIdRef>
08 </PolicySet>
```

Figure 3.2: Role Assignment <PolicySet>

<PolicySet> simply by referencing it with a <PolicySetIdReference> without need to re-write it. Figure 3.3 shows fragments of a Risk Mitigation <Policy> for the emergence example. This <Policy> consists of a <VariableDefinition> element and two <Rule> elements. Note that the <Target> element in this <Policy> is omitted, in which case it is implied by the <Target> of the Role Assignment <PolicySet>. The <VariableDefinition> (lines 02-04) is used to define a risk threshold for the mitigation strategy, which achieves splitting the risk domain $[0,1]$ into two risk intervals $[0,0.7)$ and $[0.7,1]$. Clearly, we can define an arbitrary number of such <VariableDefinition> elements (risk thresholds) to have a more fine-grained risk intervals. Specifying the risk thresholds in these variables instead of hard-coding them in the rule conditions provides a flexible way to update and maintain the risk mitigation policy. Specifically, a policy administrator only needs to change those variable definitions in order to change the risk thresholds for existing risk intervals.

Now it becomes very natural to write different rules that refer to these variables to implement a risk mitigation strategy. The first <Rule> (lines 05-21) has `Permit` as its effect when the condition is satisfied (lines 06-19); that is, the risk value for the access request lies in the interval $[0,0.7)$. Note that the <AttributeDesignator> element is used to retrieve a risk value for the access request, and the returned value must meet the specified criteria such as under the `access-risk` category and issued by a trusted authority (line 09). We describe how this mechanism works in the next section. Similarly, the second <Rule> (lines 22-28) has `Deny` as its effect if the risk value lies in the interval $[0,7,1]$. Additionally, both rules contain <ObligationExpression> (line 20 and line 27) which are evaluated into obligations by the PDP.

It can be seen that we define two or more <Rule>s in the Risk Mitigation <Policy>, each of which corresponds to checking a risk interval. For the sake of readability, we arrange these rules in an order with respect to the risk intervals from low to high ($[0,0.7)$ to $[0.7,1]$ in Figure 3.3, for example). This naturally leads us to use the `first-applicable` algorithm (OASIS, 2013, Appendix C) for combining the results of rules in the Risk Mitigation <Policy>. This algorithm forces the evaluation of the rules in the order listed in the policy, and ensures that for a particular rule, if its target and condition evaluates to `True`, then the result for the policy is the effect of the rule (`Permit` or `Deny`). In Figure 3.3, for example, if the <Rule> in lines 05-21 evaluates to `Permit`, then the second <Rule> is not evaluated, and a value of `Permit` is returned for the <Policy> (lines 01- 29).

```
00 <!-- Risk Mitigation <Policy> -->
01 <Policy PolicyId="rm:emergencydoctor:audit" RuleCombiningAlgId="first-applicable">
02   <VariableDefinition VariableId="first-risk-threshold">
03     <AttributeValue>0.7</AttributeValue>
04   </VariableDefinition>
05   <Rule RuleId="first-risk-interval-check" Effect="Permit">
06     <Condition><Apply FunctionId="function:and">
07       <Apply FunctionId="double-greater-than-or-equal">
08         <Apply FunctionId="function:double-one-and-only">
09           <AttributeDesignator Category="access-risk" AttributeId="risk" Issuer="TA"/>
10         </Apply>
11         <AttibuteValue>0</AttributeValue>
12       </Apply>
13       <Apply FunctionId="function:double-less-than">
14         <Apply FunctionId="function:double-one-and-only">
15           <AttributeDesignator Category="access-risk" AttributeId="risk" Issuer="TA"/>
16         </Apply>
17         <VariableReference VariableId="first-risk-threshold"/></VariableReference>
18       </Apply>
19     </Apply></Condition>
20     <ObligationExpressions>Log and Alert</ObligationExpressions>
21   </Rule>
22   <Rule RuleId="second-risk-interval-check" Effect="Deny">
23     <Condition><Apply FunctionId="double-greater-than-or-equal">
24       <Apply FunctionId="function:double-one-and-only">
25         <AttributeDesignator Category="access-risk" AttributeId="risk" Issuer="TA"/>
26       </Apply>
27       <VariableReference VariableId="first-risk-interval-check"></Apply></Condition>
28     <ObligationExpressions>Log</ObligationExpressions>
29   </Rule>
30 </Policy>
```

Figure 3.3: Risk Mitigation <Policy>

3.3 Risk assessment

Recall that, given an access request, a Role Assignment <PolicySet> is evaluated to Permit for the request only if the conditions defined in its target are met by the request and its associated Risk Mitigation <Policy> evaluates to Permit (which means the risk of granting this request lies in an acceptable risk interval). Let us now look at how to use XACML to compute the risk associated with an access request in more detail. As we mentioned while defining our abstract model, incurred by granting a request from a subject to access a resource could depends on various factors such as:

- The history of the subject requesting access, comprising previously authorised access requests and prior fulfillment of obligations;
- The relationship between the requesting subject and the resource (e.g. a doctor-patient assignment);
- The position of the subject in an organizational hierarchy (corresponding to the notion of role in an RBAC system);
- The purpose for subject's access request (operational need). In particular we could have to take into account also the risk derived from denying the access request; and
- The threat level of the environment in which the request was submitted...

Since XACML itself supports the use of attributes when constructing request contexts and policies, it would be natural to express these factors as attributes and choose a suitable XACML

functions to combine these attributes into a risk value in the rule condition. As a generic solution, However, the XACML predefined functions are limited; it is also not clear whether XACML accommodates the definition of an arbitrary new function, such as the complex formula used to compute risk in a multi-level security model (Cheng et al., 2007).

Instead we propose a method in which the risk calculation is conducted by a dedicated module of the PIP. As shown in the previous section, we introduce a special attribute, namely `risk`, under the `access-risk` category and require that the values for this attribute are issued by a special trusted authority. When evaluating the Risk Mitigation `<Policy>`, the PDP is instructed to request values for this risk attribute in the request context from the context hander. The context handler may retrieve this risk values from the PIP and then supply the required values into the request context. This suggests that the PIP should be able to compute the risk value at run-time when requested by the context harder, and this is compliant with the requirement of RAAC on the dynamic risk analysis.

3.4 A Java implementation of our RA-XACML model

We decided to explore the approach just defined by implementing a Java application for our medical emergency example introduced in Section 2.1.1. In the next sections we introduce Balana, an open-source implementation of the XACML reference architecture written in Java and we explain how we can write an improved PIP that supports the dynamic risk calculation for an access request.

3.4.1 Balana XACML 3.0 implementation

After the specification of the 2.0 version of the standard, Sun (now Oracle) has published an open source implementation of the XACML architecture that supports versions 1.1 and 2.0. This implementation is written in Java, an object oriented programming language developed by Sun itself. It is called *SunXACML* and its source code is available for download at the address `http://sunxacml.sourceforge.net/`. So far Oracle hasn't published any XACML implementation compliant with version 3.0 of the standard. More recently WSO2, an open-source software development company, has developed a new XACML engine based on the SunXACML implementation that provide supports for the new version of the standard. This implementation is called Balana and the source code of this implementation is available, together with some documentation and a tutorial at: `http://xacmlinfo.com`. It is still under development and a stable version of the software has not yet been released. To implement our system it would be useful to take advantage of the new features available in the version 3.0, especially of the capability to use variables in the obligation's attribute specification and to define new categories of attributes. For these reasons, after some tests and some minor bug-fixing we decided to use the Balana implementation that seems to be stable enough for our purpose.

WSO2 Balana provides an implementation of a standard PDP compatible with XACML 1.0, 1.1, 2.0 and 3.0 specifications and some interfaces to allow developers to write their own PIP and Policy repository modules. A developer who wants to build a system based on this implementation has to write the code for these modules, implementing the related interfaces defined in the Balana package:

- At least a PIP module. In particular, for every attribute that could be used in a policy, and that

is not present in the request context, there has to be one PIP module that allows the retrieval of the associated value. These modules must extend the `AttributeFinderModule` abstract class (Figure 3.4). The methods defined in Lines 01 and 04 return a boolean value that has to be `true` only if the module supports the retrieval of attributes using the `<AttributeDesignator>` and `<AttributeSelector>` modules. The method `getSupportedCategories` defined in Line 10 returns the set of String identifiers for the categories supported by the module while the method `getSupportedIds` in Line 13 returns the set of String identifiers for the attributes supported by the module. The two methods in Lines 16 and 20 are the core components of the module and are used by the PIP when it needs to retrieve an attribute value. The first of these two methods is used when the PDP has to evaluate an `<AttributeDesignator>` element while the second is used when evaluating an `<AttributeSelector>` element. Both the methods receive the request context as a parameter so they can retrieve the information necessary to identify to which particular instance the attribute request refers. For example, if the PDP needs to retrieve the email address of the subject of the request, the PIP will have to retrieve, from the request context, an identifier for the user who is requesting access. Once we have instantiated an object of a class that extends this abstract class we can load these modules into an object of the `AttributeFinder` class and pass this object as a parameter during the instantiation of a `PDP` class.

- At least a Policy finder module. These are modules that allows the system to search for a specific policy in a policy repository. The PDP could perform queries to the repository in two different ways:

  1. Searching for Policy/PolicySet that is applicable for a particular request, or
  2. Searching by its ID for a Policy/PolicySet that has been referred by another applicable policy.

  More modules can be loaded in the same policy finder, but while writing these modules and the correspondent policy set the developer must make sure that at least one but not more than one applicable policy will be found for every possible request. Every PolicyFinder module has to extend the PolicyFinderModule abstract class (Figure 3.5). The two methods `isRequestSupported` (Line 04) and `isIdReferenceSupported` (Line 07) return a boolean value that is used by the PAP to check whether a module can be used to search for a policy that matches a particular request or for a policy referred to by its id. The two `findPolicy` (Lines 13 and 16) methods are used to search for a policy in the two ways mentioned above.

3.4.2 Implementing the risk assessment module

Since the Balana implementation of the PIP is already structured in a modular way, it is straightforward for us to write a class called `RiskAssessmentModule` that extends the abstract class `AttributeFinderModule` and supports the attribute category `risk-category` and the attribute id `risk-value`. We are only interested in supporting the `<AttributeDesignator>` tag that is the one used for the risk-value retrieval in our risk mitigation policies. We want to

```
00   public abstract class AttributeFinderModule {
01      public String getIdentifier() {
02         return getClass().getName();
03      }
04      public boolean isDesignatorSupported() {
05         return false;
06      }
07      public boolean isSelectorSupported() {
08         return false;
09      }
10      public Set<String> getSupportedCategories() {
11         return null;
12      }
13      public Set getSupportedIds() {
14         return null;
15      }
16      public EvaluationResult findAttribute(URI attributeType, URI attributeId,
17      String issuer, URI category, EvaluationCtx context) {
18         return new EvaluationResult(BagAttribute.createEmptyBag(attributeType));
19      }
20      public EvaluationResult findAttribute(String contextPath, URI attributeType,
21        String contextSelector, Node root,
22        EvaluationCtx context, String xpathVersion) {
23          return new EvaluationResult(BagAttribute.createEmptyBag(attributeType));
24      }
25   }
```

Figure 3.4: AttributeFinderModule

```
00   public abstract class PolicyFinderModule {
01      public String getIdentifier() {
02         return getClass().getName();
03      }
04      public boolean isRequestSupported() {
05         return false;
06      }
07      public boolean isIdReferenceSupported() {
08         return false;
09      }
10      public abstract void init(PolicyFinder finder);
11      public void invalidateCache() {  }
12
13      public PolicyFinderResult findPolicy(EvaluationCtx context) {
14         return new PolicyFinderResult();
15      }
16      public PolicyFinderResult findPolicy(URI idReference, int type,
17        VersionConstraints constraints, PolicyMetaData parentMetaData) {
18         return new PolicyFinderResult();
19      }
20   }
```

Figure 3.5: PolicyFinderModule

implement the `findPolicy` method in a generic way, allowing a developer who is using our architecture to easily change the risk-calculation procedure without having to know how the Balana implementation works. To do so we define a RiskCalculator interface as showed in Figure 3.6.

```
00  public interface RiskCalculator {
01     double calculateRisk(String subjectId,String permissionId );
02  }
```

Figure 3.6: RiskCalculator Interface

The only method defined in this interface takes as a parameter two identifiers for subject and

permission and returns the value of the risk associated with the request. When using our implementation, a developer has to specify as a parameter an instance of this interface. Our `findPolicy` method of the `RiskAssessmentModule` will have to retrieve all the identifiers about subject and permission from the request context and return the result of the `calculateRisk` method.

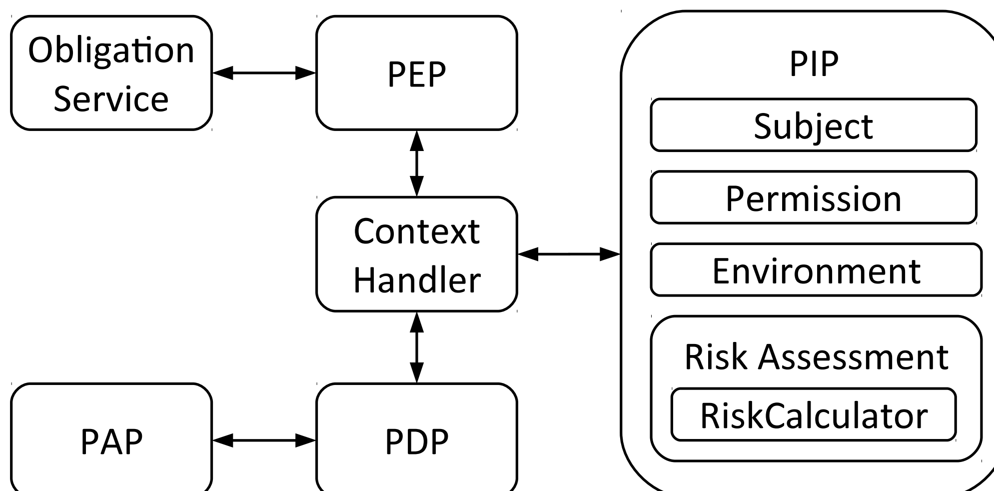

Figure 3.7: The XACML architecture with risk assessment module

In Figure 3.7 we show the architecture of our XACML-based RAAC implementation. The PIP that we implemented for our example scenario contains three modules for the retrieval of attributes of the permission, subject and environment categories. Inside the PIP we also defined the `RiskAssessment` module that refers to a RiskCalculator interface for the dynamic calculation of risk. In doing so, we are able to make our XACML policies become risk-aware without significant changes in the reference XACML architecture. In fact, the risk calculation is seen from the PDP as a normal attribute retrieval and we can write policies in which the decision about an access request is based on the actual value of the risk. The way in which the risk value is calculated depends on the particular implementation of RiskCalculator interface.

3.5 Extending the model and its XACML implementation

3.5.1 Supporting user obligations

Recall that in a RAAC system, after granting a risky access, it could be useful to apply some risk mitigation methods in order to reduce and account for the risk to which the system is exposed and that one of the most common forms of risk mitigation methods, to which we refer as user obligations, consists in obliging a subject to perform some kind of actions. Unlike system obligations, user obligations may go unfulfilled, simply because the user has either deliberately failed or forgotten to take obliged actions. Thus the system needs to define a mechanism that provide incentives for users to fulfil their obligations and punishments for those who fail or refuse to do so. In the next section we introduce an extended model based on the meta-model defined in Section 2.1.1 in which we introduce support for the definition of user obligations in our risk mitigation strategies. This extended model is based on the work by Chen et al. (2012) and it represents an instantiation of the abstract RAAC model defined there, in which we introduce only some minor

changes. Specifically, we define, for every user, a value called budget that represents, at any time, the likelihood and the capability of the given user to fulfil his obligations. For every user, the budget value will be updated according to the state of the current and previous obligations assigned to him. The decision whether or not to allow a risky access for a subject, will depend not only on the risk value associated with the request, but also on the budget value of the user. We then show how we can extend the implementation defined in Section 3.4.2 to support our extended module, introducing the definition of user obligations and budget value in the risk mitigation policies. At an enforcement level, the system has to be able to monitor the state of user obligations and undertake appropriate actions in case of failure to fulfil them.

3.5.2 Formal model

Formally, we define a function $b : S \to [0,1]$, where $b(s)$ denotes the amount of *budget* for subject s. The budget of s is used to pay deposits for granting risky accesses, and these deposits will only be returned if s fulfils obligations resulting from her granted accesses. We model a user obligation o^u as a pair $\langle A, l \rangle$, where A is a set of actions and l is a symbolic temporal interval (for example, Instantly, 10Hours) during which A must be performed. We write $\perp$ to denote the "null" obligation; the requesting user or PEP is not required to do anything for the null obligation. We also write $d \in [0,1]$ to denote an amount of deposit that the requesting user is required to pay for some risky access to be granted. Then we extend a *risk mitigation strategy* to be a list $[\langle k_0, o^s_0, \perp, 0\rangle, \langle k_1, o^s_1, o^u_1, d_1\rangle, \ldots, \langle k_{n-1}, o^s_{n-1}, o^u_{n-1}, d_{n-1}\rangle, \langle k_n, o^s_n, \perp, 0\rangle]$, where $0 = k_0 < k_1 < \cdots < k_n \leqslant 1$, $o^s_i \in O^s$, $o^u_i \in O^u$ and $0 \leqslant d_1 < \cdots < d_{n-1} \leqslant 1$. Given a request $q = \langle s, r, c\rangle$ and a system state σ, we define an extended authorisation function Auth_e as,

$$\mathsf{Auth}_e(q,\sigma) = \begin{cases} \langle \mathsf{allow}, o^s_0, \perp, 0\rangle & \text{if } \mathsf{Risk}(q,\sigma) < k_1, \\ \mathsf{Bud}(q, o^s_i, o^u_i, d_i) & \text{if } \mathsf{Risk}(q,\sigma) \in [k_i, k_{i+1}),\ 1 \leqslant i < n, \\ \langle \mathsf{deny}, o^s_n, \perp, 0\rangle & \text{otherwise,} \end{cases}$$

where the budget checker function Bud is defined as,

$$\mathsf{Bud}(q, o^s_i, o^u_i, d_i) = \begin{cases} \langle \mathsf{allow}, o^s_i, o^u_i, d_i\rangle & \text{if } b(u) \geqslant d_i, \\ \langle \mathsf{deny}, o^s_i, \perp, d_i\rangle & \text{otherwise.} \end{cases}$$

Informally, the semantics for how Auth_e works is very close to Auth as we explained in Sect 2.1.1, so we omit its explanation here. However, it would be important to see the basic ideas behind the Bud function. Essentially, specifying deposits in a risk mitigation strategy provides a means of sanctioning subjects, as well as restricting the number of risky accesses to be granted to subjects. Specifically, if subject s has sufficient budget to pay deposit d_i, then d_i will be deduced from s's budget when u accepts o^u_i and d_i is returned when o^u_i is fulfilled. If s's budget is less than the required deposit d_i because of failures to fulfil prior obligations, or the outstanding obligations to be fulfilled, then s is denied access to resource, even though the risk of allowing the access would be acceptable.

There are a few operations that the PEP is required to execute when receiving a response from the Auth_e function. In fact, given a request $\langle s, r, c\rangle$, the actions that the PEP has to perform vary

depending on one of four possible results (shown below) being returned.

- $\langle \mathsf{allow}, o_i^s, \perp, 0\rangle$ and $\langle \mathsf{deny}, o_i^s, \perp, 0\rangle$: The PEP is required to perform system obligation o_i^s in conjunction with enforcing an authorisation decision (either "allow" or "deny").
- $\langle \mathsf{allow}, o_i^s, o_i^u, d_i\rangle$: In addition to enforcing the "allow" decision and executing o_i^s, when s accepts o_i^u, the PEP would be able to transform o_i^u into a more concrete form whose execution the system can monitor. At the same time, the PEP needs to deduct the deposit d_i from s's budget and return it when o_i^u is fulfilled.
- $\langle \mathsf{deny}, o_i^s, \perp, d_i\rangle$: In addition to enforcing the "deny" decision and executing o_i^s, the PEP needs to show a message dictating that s's budget is not sufficient to pay deposit d_i for gaining this access. This is to inform s to seek a way to increase her budget, since the risk of allowing her to access r is acceptable.

In the next section, we illustrate how to extend the XACML architecture to support these operational semantics of PEP.

3.5.3 XACML implementation

To implement the extended model using XACML, we first look at how to extend the Risk Mitigation `<Policy>` to include the concept of budget. We consider the example scenario introduced above and define a risk mitigation strategy for the permission to read Alice's SCR. After that we show how to write a risk mitigation `<Policy>` that implements this mitigation strategy explaining which improvements have to be introduced, with respect to the XACML architecture defined in 3.4.2 to support the additional features of our model.

The risk mitigation strategy for the permission to read Alice's SCR is represented in Figure 3.8. The horizontal bar represents the different risk-value ranges. When the risk-value lies in the white area (below 0.2) the access is allowed without any requirement for the budget value and any returned obligation. When it lies in the grey area, between 0.2 and 0.7, a budget of 0.3 is required and the user is obliged to send an email to justify the access within one day and two hours from the time when the request has been accepted. When the risk value is above 0.7 (black area) the access request is always denied, no matter how high is the budget value of the subject.

Recall that the Risk Mitigation `<Policy>` includes `<Rule>`s, each of which is used to check whether the risk lies in a specific interval. For some rules that have a `<Permit>` effect (which means the risk is acceptable), we now need to further split each of them into two cases: whether the requesting subject has sufficient budget or not. In order to provide greater flexibility and economy in writing such complex rules, we take an approach that, for every risk interval, two boolean variables are defined: one, called risk-check, will be evaluated to true when the risk-value lies in the given interval, the other, called budget-check, will be evaluated to true when the budget value of a user is above the the budget threshold defined for that interval. These boolean variables can be referenced many times when defining rules.

The risk mitigation `<Policy>` that implements the risk mitigation strategy associated to the permission to read Alice SCR is showed in Figure 3.9. The `<VariableDefinition>`s (lines 01-02) define the two risk thresholds that form three risk intervals: $[0, 0.2)$,$[0.2, 0.7)$ and $[0.7, 1]$. The `<VariableDefinition>` (line 03) introduces a budget value 0.3 that corresponds

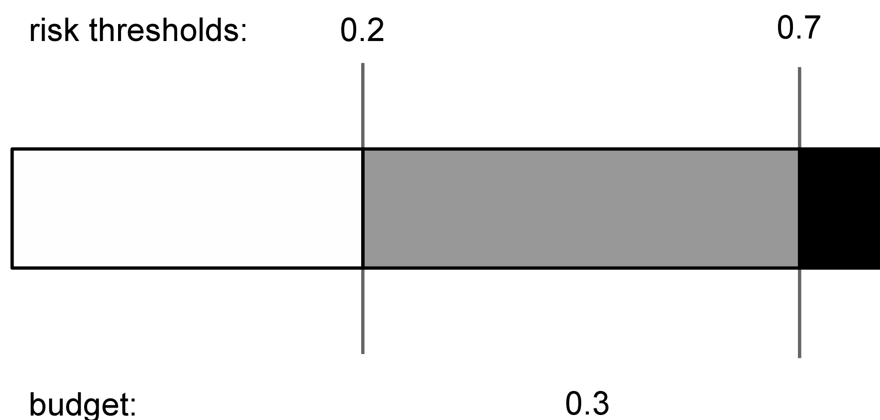

Figure 3.8: Risk mitigation strategy for permission to read Alice's SCR

to the risk interval $[0.2, 0.7)$. In the extended model, there is no budget associated with the least and most risky intervals ($[0, 0.2)$ and $[0.7, 1)$, for example), hence we simply omit them here. The `<VariableDefinition>`s (lines 04-25) define three boolean variables, each of which holds a `True` value if the risk value for the request lies in the corresponding interval and `False` otherwise. Similarly, the boolean value in the `<VariableDefinition>` (line 26) indicates whether the requesting subject's budget is greater than 0.3. The budget of the subject is treated as an attribute, and its value is retrieved by the means of `<AttributeDesignator>` with the `access-subject` category defined in the policy. The policy consists of four rules, each of which just simply include one reference to the correspondent risk-check variable and one to the budget-check variable. For example, the second rule (lines 30-49) is applicable (evaluates to `Permit`) if its referred two `<VariableDefinition>` elements (lines 24-26) are both `True`.

XACML allows every `<Rule>` to include a set of `<ObligationExpression>` elements, each of which represents an obligation. An `<ObligationExpression>` can include an arbitrary number of attribute assignments, which forms the *arguments* of the action defined by the obligation. For example, the obligation expression with `user:email` id (lines 39-49) defines three attributes: the first attribute (lines 40-42) specifies who is obliged to fulfil the obligation; the second (lines 43-45) indicates the email address of the requester's line manager; and the third (lines 26-28) defines the temporal interval (1 day and 2 hours) during which the requester has to perform the obligation. Note that the `<ObligationId>` attribute (line 19) identifies the action; that is, oblige the requester to send an email. When evaluating this `<ObligationExpression>`, the PDP determines the values for `<requester>` and `<emailID>` at runtime by the means of an `<AttributeDesignator>`, and sends the resulting obligation to the PEP in the response context. XACML specification states that the `<ObligationExpression>` element is used to specify obligations that have to be fulfilled by the PEP. As stated in the XACML specification, the PEP itself has to understand and know how to handle the obligation when receiving the response. Since the PEP is application dependent we are free to define our own syntax and semantics for an obligation definition. To this end, we simply distinguish between a system obligation and a user

```
00 <Policy PolicyId="crm:emergencydoctor" RuleCombiningAlgId="first-applicable">
01  <VariableDefinition VariableId="risk-threshold-1">0.2</VariableDefinition>
02  <VariableDefinition VariableId="risk-threshold-2">0.7</VariableDefinition>
03  <VariableDefinition VariableId="budget-threshold-1">0.3</VariableDefinition>
04  <VariableDefinition VariableId="risk-check-1">
05   <Apply functionId=":1.0:function:and">
06    <Apply FunctionId="double-greater-than-or-equal">
07     <Apply FunctionId="double-one-and-only">
08      <AttributeDesignator MustBePresent="false"
09       Category="risk"
10       AttributeId="risk-value"
11       DataType="double" />
12     </Apply>
13     <VariableReference VariableId="risk-0"></VariableReference>
14    </Apply>
15    <Apply FunctionId=":1.0:function:double-less-than">
16     <Apply FunctionId=":1.0:function:double-one-and-only">
17      <AttributeDesignator MustBePresent="false"
18       Category="risk"
19       AttributeId="risk-value" DataType="double"/>
20     </Apply>
21     <VariableReference VariableId="risk-1"></VariableReference>
22   </Apply></Apply>
23  </VariableDefinition>
24  <VariableDefinition VariableId="risk-check-2">?(0.2<=risk<0.7)</VariableDefinition>
25  <VariableDefinition VariableId="risk-check-3">?(0.7<=risk<=1)</VariableDefinition>
26  <VariableDefinition VariableId="budget-check-1">?(budget>=0.3)</VariableDefinition>
27  <Rule RuleId="first-risk-interval" Effect="Permit">
28   <Condition><VariableReference VariableId="risk-check-1"></Condition>
29  </Rule>
30  <Rule RuleId="second-risk-interval-suffbudget" Effect="Permit">
31   <Condition><Apply FunctionId="function:and">
32    <VariableReference VariableId="risk-check-2"></VariableReference>
33    <VariableReference VariableId="budget-check-1"></VariableReference>
34   </Apply></Condition>
35   <ObligationExpressions>
36    <ObligationExpression ObligationId="system:deduct-budget">
37      ...</ObligationExpression>
38    <ObligationExpression ObligationId="user:email">
39     <AttributeAssignmentExpression AttributeId="requester">
40      <AttributeDesignator AttributeId="subject-id" Category="access-subject"/>
41     </AttributeAssignmentExpression>
42     <AttributeAssignmentExpression AttributeId="emailId">
43      <AttributeDesignator AttributeId="manager-email" Category="access-subject"/>
44     </AttributeAssignmentExpression>
45     <AttributeAssignmentExpression AttributeId="duration">
46      <AttributeValue DataType="#dayTimeDuration">P1DT2H</AttributeValue>
47     </AttributeAssignmentExpression>
48    </ObligationExpression></ObligationExpressions>
49  </Rule>
50  <Rule RuleId="second-risk-interval-insuffbudget" Effect="Deny">
51   <Condition><Apply FunctionId="function:and">
52    <VariableReference VariableId="risk-check-2"></VariableReference>
53    <Apply FunctionId="function:not">
54     <VariableReference VariableId="budget-check-1"></Apply>
55   </Apply></Condition>
56   <ObligationExpression ObligationId="system:inform-budget">
57       ...</ObligationExpression>
58   </Rule>
59   <Rule RuleId="third-risk-interval" Effect="Deny">
60    <Condition><VariableReference VariableId="risk-check-3"/></Condition>
61  </Rule></Policy>
```

Figure 3.9: Complex Risk Mitigation <Policy>

obligation by naming them differently in <ObligationId>s (see Figure 3.9, lines 38-39), and leave the PEP to interpret the difference and handle them differently.

3.5.4 Implementing the obligations service

We now propose an extensible structure of the PEP to support obligation monitoring and enforcement as shown in Figure 3.10. We describe how this structure manages the different kinds of obligations returned by the PDP after an access request has been evaluated in order to comply with the previously defined extended model. When PEP receives a response that contains an authorisation decision and a list of obligations (system and user obligations), it forwards the obligation set to the *obligation handler* before enforcing the decision (step 1). The obligation handler classifies these obligations into two sets: system obligations and user obligations, and forwards system obligations to the *obligation enforcement point* (step 2). The obligation enforcement point performs the system obligations by invoking corresponding *fulfillment module* inside. This fulfillment modules are responsible to understand and correctly enforce the obligations, each of which is designed to support a particular type of obligation actions. For example, one module is to handle the deduction of a deposit for a requester's budget, and the other is to execute logging and auditing. This modular structure allow us to maintain the basic obligation handling structure, adding the support for new obligations when needed. Once these system obligations are successfully performed, the PEP will be notified so as to proceed with the decision enforcement[1] (steps 3-4b). Meanwhile, the obligation handler forwards user obligations to the *fulfillment monitor* (step 4c), which makes these obligations switch into active states according to their temporal constraints and current system clock. Additionally, the fulfillment monitor maintains the list of pending user obligations. It checks whether any pending obligations are fulfilled by a user-initiated action, and if any have fulfilled, it notifies the obligation handler. It also monitors whether any pending obligations that have deadlines that have been passed without appropriate actions occurred, and hence it notifies the obligation handler for those violated obligations (steps 5-6). Like the obligation enforcement point, the fulfillment monitor consists of a number of *monitor modules*, each of which is implemented to be able to monitor a particular type of user obligations.

In order to implement budget control in our extended model, we instruct the obligation handler, for each corresponding decision, to process the list of obligations in an order $\langle o_0^s, o_1^s, \ldots, o_i^u, o_j^u, \ldots, o_n^s \rangle$, where o_0^s is the system action that deducts deposit from the requester's budget, while o_n^s is the system action that returns the deducted budget back to the requester's account. It means that o_n^s will be only executed if the previous user obligations $\langle o_i^u, \ldots, o_{n-1}^u \rangle$ are successfully fulfilled.

In short, we propose an architectural structure for implementing the obligations service component defined in the XACML standard. Our proposed approach supports the enforcement of both system and user obligations in a generic manner without changing the standard XACML architecture and language.

3.6 Discussion

There is a considerable body of work on risk-aware access control, much of it focusing on developing models for incorporating risk in multi-level security (Cheng et al., 2007; Ni et al., 2010) and role-based access control (Bijon et al., 2012; Chen and Crampton, 2011). Very little of that

[1] This is consistent with the way defined in the XACML specification; that is, an authorisation decision can only be enforced if the PEP can understand and discharge the returned obligations (OASIS, 2013, Section 2)

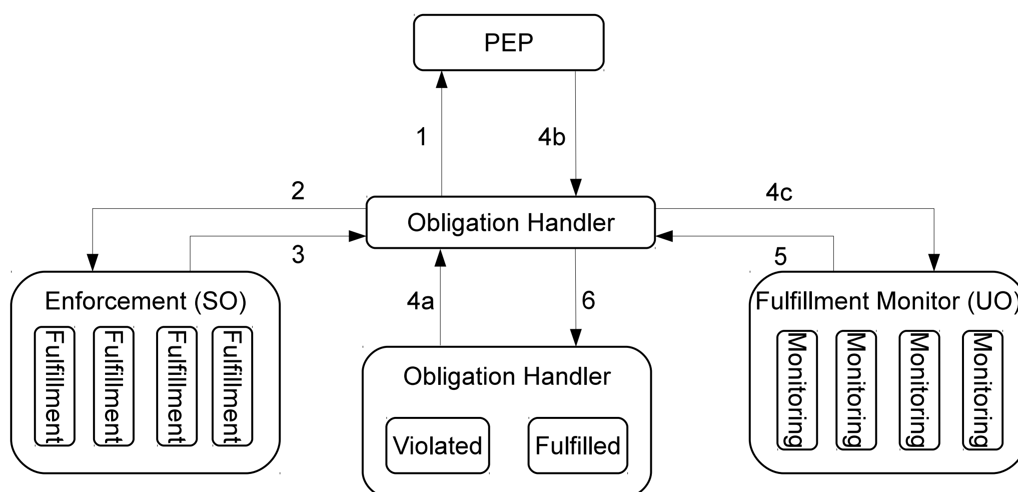

Figure 3.10: The extended architecture for supporting obligations

research is concerned with the authorisation architectural design that accommodates the awareness of risk, with the exception of the work of Chen et al. (2010). The latter work extends XACML with new XML languages and functional components to support risk-adaptive access control. In contrast, our approach to implementing RAAC is fully compliant with the XACML standard without introducing extra elements.

Baracaldo and Joshi (2012) propose a model that includes risk and trust in RBAC systems in order to react to anomalous behaviour and insider threats. The model defines, for every user, a trust value that is based on the history of the user and the context of the system. Every user can request to temporally activate some roles in order to access to the related permission set. At every request is associated a risk value that should include the imminent risk associated with the permissions acquired through the roles, and the risk due to inference of unauthorized objects. The model supports generic risk mitigation mechanism but doesn't specify any possible mechanism and thus doesn't provide any user obligation monitoring system. Moreover the model is built upon the RBAC model and allows the assignment of permissions to the user only by means of a role activation. We take a more generic approach that associates a risk mitigation strategy to a generic permission that can be singularly activated.

The XACML standard treats a system obligation as an attribute assignment, and leaves the interpretation of these obligations to the PEP. Further, it does not provide support for user obligations. In the XACML technical committee, there is some work, called "Obligation Families" (OASIS, 2007), which attempts to define additional mechanisms for obligation processing and enforcement, but this is very preliminary and is not reflected in the current XACML standard. Additionally, Li et al. (2012) recently introduced a comprehensive specification and processing model for obligations by extending XACML specification and architecture. Certainly this work is complementary to our approach to handling obligations, but we take a different view that, for someone who is committed to existing XACML architecture, it would be straightforward to use our approach for supporting user obligations and incorporating the notion of risk.

Chapter 4

Risk-Aware GBAC

4.1 Running example

In this chapter we introduce a risk-aware group-based access control (RA-GBAC) model as a way to promote optimal permissions allocation and sharing, on the basis of the already mentioned *least-privilege principle*, in a dynamic collaborative environment. In order to illustrate the motivations that led us to the definition of this RA-GBAC model and the basic requirements that this model has to satisfy we introduce an example scenario that will show how this model works and how it can be useful for managing some collaboration scenarios, especially in an emergency situation.

After a car accident, Alice is taken by an ambulance to the nearest hospital. The hospital is facing an emergency situation and most of the clinical staff members are already busy in other emergency interventions. Bob is a *surgeon* who is working in the emergency room of the hospital. Carol is a *nurse trainee* and at the moment is the only nurse that is not busy in other interventions. Charlie is an *anaesthetist trainee*, and is the only anaesthetist not busy in other interventions. The relevant part of the role hierarchy of this hospital is showed in Figure 4.1. A *nurse* and an *anaesthetist* can access to all of the permissions associated respectively with the *nurse trainee* and the *anaesthetist trainee* role. A *nurse trainee* and a *anaesthetist trainee* can access respectively only to some of the *nurse* and *anaesthetist* permissions. Every employee of the hospital act empowering one or more roles and can use his risk budget to temporally activate other roles using the risk-aware RBAC model defined in Section 2.1.1.

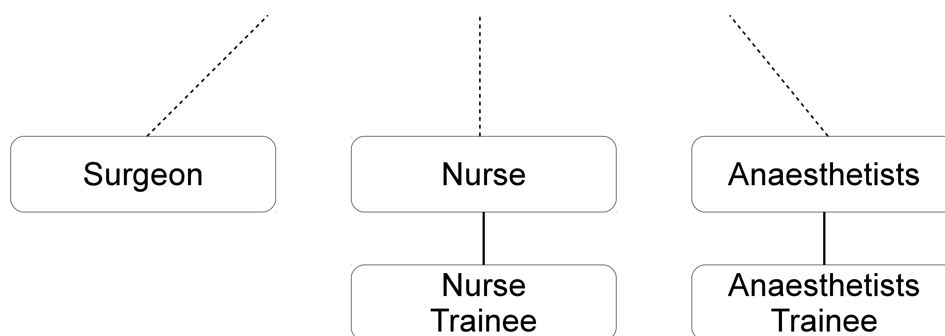

Figure 4.1: Scenario role hierarchy

After a quick visit, Carol determines that Alice needs to be brought to the emergency room for a surgical operation. To operate, Bob needs the assistance of a nurse and an anaesthetist but

there are not any one available other than, respectively, Carol and Charlie. Carol would like to temporally activate the *nurse* role. She sends a request to the access control system. The system assess a risk-value for the request that lies in a middle range but she doesn't have the amount of budget defined by the risk mitigation strategy for granting an access in that risk-range. Charlie as well would like to temporally activate the *anaesthetist* role, but he doesn't have the budget value defined for his risk range. It could be reasonable in this situation for Bob, Carol and Charlie to form a team in order to share the responsibilities for the risk incurred by allowing Carol to activate the nurse role and Charlie to activate the anaesthetist role. In particular Carol and Charlie could use some of the Bob's budget to temporally activate their roles. This would be an advantage for Carol and Charlie, that could access to some permission using Bob's budget. Bob would be responsible for both Carol and Charlie, and could lose part of his budget if they misuse their permission or don't fulfil some of the resulting obligations but, doing so, he could proceed with the surgical operation.

4.2 Motivations and requirements

The example from the previous section led us to the conclusion that there is need for the definition of an access control framework that promotes the sharing of permissions between a collective of users that collaborate in a system for the fulfilment of a common goal. In particular, it would be reasonable to allow a user, that acting as an individual would not be granted a given permission, to access to that permission in behalf of a group of users in which he acts together with some more trustworthy users. In this way, the members of the group would share, not only the access permissions, but also the responsibilities in case of misuse of those permissions. When considering the RAAC model defined in Section 3.5.1 this means that all the members of the group will be responsible for the fulfilment of the obligations returned when granting an access. This situation could result in some risk for the users with greater trustworthiness in the group since they could result co-responsible for the less trustworthy users misusing the shared permissions. This risk could be considered acceptable by these users in those situation when they need to delegate some important task that, for some reasons, they cannot fulfil on their own and for which the needed permissions would not normally be granted to a less trustworthy user because of their budget value being too low.

To develop a generic model that can be used for managing both collaborative and individual access requests, always in respect of some reasonable security boundaries, we identified some requirements. Recall that, in our individual RAAC abstract model, when the risk value is in the last range of a risk mitigation strategy, the returned decision will always be deny, no matter how high is the user's budget. If the risk for any member of the team accessing individually at a given permission lies on the last range, we want any access request from the team for the given permission to be denied. Another requirement states that, when evaluating an access request for a team which have only one member, the decision has to be equivalent to the one the model defined in Section 3.5.1 would have returned for the given member acting as an individual. Doing so we don't need to implement two different models, one for individual and one for collaborative requests. We just need to define one singleton group for every user, and use these groups when we want to send a request for an individual access.

Recall that, in our example scenario (Section 4.1) all the permissions refer to the temporary activation of a role to which a subject would not normally be allowed and that, in order to fulfil its job, the team of the example needs both the permissions sets associated to the activation of the nurse and the anaesthetist role. Consider the case in which the team asks for the two permissions once at a time and, while the request for activating the nurse role is granted, the one for the anaesthetist role is denied because the group members cannot provide the needed budget or because of a too high risk level. In this situation it would be useless, and so even dangerous for the system, to allow the team to activate the nurse role. For this reason we introduce in our model a complex permission that is, in fact, a set of permission that can be activated with a single request. Moreover, since when a risky permission is granted to a team, all the members of the team will be considered responsible for the permissions eventually being misused, it is reasonable to allow all these users to benefit from the allowed access to this permission. This means that all the users that are members of the team will have to be able to execute the risky permissions. In our example this means that every user of the team is allowed to activate all the roles associated with the task. It is not reasonable to require that all the users of one team that want to use the shared permissions have to do this in the same instant of time. For this reason, we introduce the concept of task as a set of permission associated to a time duration, during which, starting from the activation time-instant, the task is considered valid. All the users of the team can access to all the permission associated with the task during this whole time-interval.

While we are interested in monitoring the fulfilment of the obligations by the users, it is out of the scope of an access control system to monitor whether a task or not will be correctly fulfilled by a group of users. Our goal in fact is only to provide a user, or a group of users with all, but no more of the permissions they need to perform their job tasks.

4.3 Formal model

4.3.1 Integrating tasks into RAAC

There has been some work on *task-based authorisation* (Irwin et al., 2008; Thomas and Sandhu, 1997), where tasks are introduced as a central concept that associates users to permissions. Specifically, in task-based systems, a task is treated as a first-class entity that is associated with a set of permissions that the system deems to be necessary to fulfil the task. In addition, a task is often associated with a temporal constraint, indicating when it should be finished. This way of modelling tasks provides a simple and natural way to achieve the *least-privilege* principle (Saltzer and Schroeder, 1975) by restricting a set of permissions that is required for the task, and making those permissions only available during the task fulfilment.

When a task arises in a system, the task may be assigned to a single user or to a group of users who collaboratively fulfil it. That is, only when users are assigned to a task, these users can use the permissions associated with the task. In general, the assignment of tasks to users can be done in two different ways. The system may have a means of assigning tasks to the most appropriate users, and subsequently these users are obliged to perform the tasks. Alternatively, in a more flexible setting, users initiate requests to perform tasks in order to gain benefits of accomplishing the tasks. In both ways, the system must incorporate a module, which we call the *authorisation decision function* (ADF), that can ensure that the user(s) to whom a task is assigned, is authorised

to perform the task.

In most existing approaches, the ADF determines whether a user is permitted to perform a task according to some predefined authorisation policies. However, in many situations, those policies are found too rigid, which results in some important task being unable to be carried out, even though some suitable users are available. In this chapter, we propose a risk-aware authorisation mechanism that determines whether to allocate a task to a set of users based on the risk of those users to misuse the permissions associated with the task. Hereafter, we define a risk-aware model following the way where users initialise requests to perform tasks, since the design of our risk-aware model can be readily modified to accommodate the static assignment of users to tasks by the system.

4.3.2 Constructing a reference monitor

Let G be a set of groups or teams, and $UG \subseteq U \times G$ be a user-group assignment relation. Given $g \in G$, we write $\mathsf{Users}(g)$ for the set of users to which the group g is explicitly assigned by the UG relation; that is, $\mathsf{Users}(g) = \{u \in U : (u,g) \in UG\}$. Given $g \in G$, we say that $\mathsf{Users}(g)$ is a set of members of g. For the sake of completeness, for all $u \in U$, we assume there exists a corresponding $g \in G$, where $\mathsf{Users}(g) = \{u\}$. Given $g_1, g_2 \in G$, we define $g_1 \leqslant g_2$ if and only if $\mathsf{Users}(g_1) \subseteq \mathsf{Users}(g_2)$. Clearly, $\leqslant$ is a partial order on G, and we write $\langle G, \leqslant \rangle$ to denote the partial ordered set of groups G. In this section, we do not specify the authorisation semantics in terms of hierarchical organization of groups, but we will discuss the benefits for such group organization in Chapter 5.

Similarly, we define a budget for each group in the system. Given $g \in G$, we write $b(g) \in [0,1]$ to denote the budget of g; the initial value of $b(g)$ is defined to be 0 for all g. When members of group g decide to initial a request to perform a particular task, they have to provide the system with enough budget. There are different way in which we can decide how much budget every user has to put in order to reach the needed value. In this section we only introduce two different example mechanisms which we discuss in more detail in Chapter 5. The total available team-budget could be calculated by the system as the sum of all the budget values of the members. In this case, the system has to decide, when the permission of activating a task is granted, which is the amount of budget that every user has to put. We discuss some sharing mechanism for this case in Chapter 5. Another possibility is to define, for every team, an attribute called `team-budget`. Every user, at any moment, can top-up the team-budget using part of his individual budget. A request is then granted only if the team-budget is higher than the required value. This mechanism is not transparent to the users and requires them to manage their budget values. On the other hand it allows the different members to negotiate the budget deposit that everyone has to use to join a team in requesting an access and thus how to share the responsibilities incurred by requesting to fulfil a task. For the basic model now, we assume that all members contribute their budgets equally regardless the relative seniority of the members in the group. When a user u logins in the system, u can either request to invoke permissions or to perform tasks. When u requests to perform a particular task, u has to choose which group to be the entity to which authorisation of performing the task is evaluated. Given a user $u \in U$, we write $\mathsf{Grps}(u)$ for the set of groups to which a user u is explicitly assigned by the UG relation; that is $\mathsf{Grps}(u) = \{g \in G : (u,g) \in UG\}$. Clearly, u is only allowed to activate a group g such that $g \in \mathsf{Grps}(u)$, and all authorisation information associated

with g are passed into the ADF to determine whether g is authorised to perform the task.

Let K be a set of tasks, and P be a set of permissions. We introduce a permission-task assignment relation $PK \subseteq P \times K$. Given $k \in K$, we write $\mathsf{Prms}(k)$ for the set of permissions to which the task k is explicitly assigned by the PK relation; that is, $\mathsf{Prms}(k) = \{p \in p : (p,k) \in PK\}$. In other words, $\mathsf{Prms}(k)$ is a set of permissions that is required to perform the task. We model an collective obligation o^g as a pair $\langle A, l\rangle$, where A is a set of actions and l is a symbolic temporal interval during which A must be performed by one of the members of g. All the members of the group are to be considered responsible for a failure in fulfilling the obligation before the deadline. System obligations o^s are defined in the same way as in the RAAC model described in Section 2.1.1. We write $\perp$ to denote the "null" obligation; the requesting user or PEP is not required to do anything for the null obligation. We define O^s and O^g respectively as a set of system and collective obligations. We then associate each task k with a risk mitigation strategy, denoted by $\mu(k)$. Clearly, the definition of the risk mitigation strategy is determined by all the permissions associated with task k. We introduce a function $\lambda : K \to L$, where $\lambda(k)$ denotes the symbolic temporal interval during which the task k should be completed. We also introduce a group-task assignment relation $GK_T \subseteq G \times K \times \mathcal{T}$, where $(g,k,T) \in GK_T$ means that the assignment of task k to group g is enabled for all points $t \in T$. When k is granted to be performed by a particular group g, $\lambda(k)$ is transformed into a concrete time interval $T = [t_s, t_e]$, and a tuple $\langle g,k,T\rangle$ is appended into GK_T. Let t_i be an instant of time, C be a set of variables $\langle UG, GK_T, PK, \mu\rangle$ that represent the state of the system and $C(t_i)$ be a values assignment, for the variable in C, at the instant of time t_i. We now present how the authorisation decision function determines whether to grant a group the permission to perform a task.

We define a risk function $\mathsf{Risk_g} : G \times K \times C(t_i) \to [0,1]$ that computes the risk of granting a group to perform a task. We model a *group* request as a tuple $\langle g,k,t_i\rangle$, where $g \in G$ is a group and $k \in K$ is a task and t_i represents the instant of time in which the request is sent. Given a group request $\langle g,k,t_i\rangle$, we believe that, depending on the context $C(t_i)$ there are a number of different ways to compute $\mathsf{Risk_g}(g,k,C(t_i))$ as a function of $\mathsf{Risk}(u,k,C(t_i))$ for all $u \in \mathsf{Users}(g)$. Before discussing different possibilities of computing $\mathsf{Risk_g}(g,k,C(t_i))$, let us introduce the authorisation decision function $\mathsf{Auth_g}$. Given a group request $\langle g,k,C(t_i)\rangle$ we define an authorisation function $\mathsf{Auth_g}$ as:

$$\mathsf{Auth_g}(\langle g,k,t_i\rangle) = \begin{cases} \langle \mathsf{allow}, \emptyset, 0\rangle & \text{if } \mathsf{Risk_g}(g,k,C(t_i)) < k_1, \\ \mathsf{Bud_g}(\langle g,k\rangle, O_i^s, O_i^g, d_i) & \text{if } \mathsf{Risk_g}(g,k,C(t_i)) \in [k_i, k_{i+1}),\ 1 \leqslant i < n, \\ \langle \mathsf{deny}, \emptyset, 0\rangle & \text{otherwise,} \end{cases} \tag{4.1}$$

where the budget checker function $\mathsf{Bud_g}$ is defined as,

$$\mathsf{Bud_g}(\langle g,k\rangle, O_i^s, O_i^g, d_i) = \begin{cases} \langle \mathsf{allow}, O_i^s, O_i^g, |\mathsf{Users}(g)| * d_i\rangle & \text{if } b(g) \geqslant |\mathsf{Users}(g)| * d_i, \\ \langle \mathsf{deny}, \perp, \perp, |\mathsf{Users}(g)| * d_i\rangle & \text{otherwise.} \end{cases} \tag{4.2}$$

Note that, RAAC can be seen as a special case of $\mathsf{RAAC_g}$ in which $|\mathsf{Users}(g)| = 1$ for all

$g \in G$. In fact, we can interpret a group request $\langle g,k \rangle$ as a set of individual requests $\langle u,k \rangle$ for all $u \in \mathsf{Users}(g)$. We then determine the risk of granting a group request by combining the risk of granting individual requests. We believe that there are at least two ways to compute the risk of granting a group request, which may result in different types of obligation assigned to a group and different amounts of budget a group need to put together. Given a group request $\langle g,k \rangle$, one possibility is to compute the risk of granting this request to be

$$Risk(t,k) = \max\{\mathsf{Risk}(u,k) : u \in \mathsf{Users}(g)\}. \tag{4.3}$$

In other words, the risk of granting $\langle g,k \rangle$ is determined by the maximum value in the set comprising the risk of all individual accesses. This is a conservative approach in a sense that the risk of a group request is determined by the least trustworthy individual in the group, thereby all members in the group are collectively responsible to perform a more restrictive obligation. In addition, on the basis of equally share of budget among the members of a group, some members in the group actually pay more budget than the one they need on individual accesses. An alternative way of computing the risk associated with a group request $\langle g,k \rangle$ is to compute

$$\frac{\sum \mathsf{Risk}(u,k) : u \in \mathsf{Users}(g)}{|\mathsf{Users}(g)|}.$$

This computation essentially takes the average of risks associated with individual accesses from all members in the group. In this case, a low trustworthy user is benefit for being in the group, because she may be able to gain access to some resources by associating with high trustworthy users in the group.

From the point of view of safety, we may wish to prevent a group from performing a task if there exists a group member whose individual request to this task is too risky to be granted. Therefore it may be desirable to impose a constraint on the group request, that is, for all $u \in \mathsf{Users}(g)$, $\mathsf{Risk}(u,k) < k_n$, where k_n is most risky threshold defined in the risk mitigation strategy associated with k. In this case, for some users who have ran out budget due to unfulfilled obligations, they can still request to perform tasks by presenting a group with other members putting individual budgets into the group budget. Of course, it may be appropriate to compute the risk of a group access as a more complex function of the risks associated with individual accesses, which will be discussed in Chapter 5

We now introduce an authorisation function that determines whether users are authorised to perform permissions. Given a request at time t, denoted by $\langle u,p,t \rangle$, and a $\mathrm{RAAC_g}$ state $C(t_i)$, we define an authorisation decision function $\mathsf{Auth_u}$ that returns allow if there exists g,k such that $\langle u,g \rangle \in UG$, $\langle g,k,t \rangle \in GK_T$ and $\langle p,k \rangle \in PK$; and deny (otherwise).

4.3.3 Enforcing authorisation decisions

There are a few actions that the policy enforcement point (PEP) is required to perform when receiving an allow decision from the authorisation function $\mathsf{Auth_g}$. Firstly, when an authorisation decision regarding $\langle g,k \rangle$ along with the corresponding obligations O_i^s, O_i^g arrive at the PEP, it tries to fulfil O_i^s and, only if this is possible, it enforces the decision and transforms the obligations in O_i^g into a more concrete form whose execution the system can monitor. At the same time, the PEP

needs to deduct the deposit $|\mathsf{Users}(g)| * d_i$ from the group budget $b(g)$. When all the obligations in O_i^g are fulfilled by one of the members in g, d_i is returned to all members $u \in \mathsf{Users}(g)$ in case that the equally sharing mechanism is adopted. We will discuss the other possible mechanisms for sharing budgets among group members in Chapter 5.

As we mentioned, when a request $\langle g,k \rangle$ is granted, the PEP should also explicitly assign the task k to the granted group g, so that only members of g can carry out the task via the invocation of permissions associated with the task.[1] Specifically, suppose a request from g to perform k is allowed and $\lambda(k) = l$, the PEP transforms l into a concrete time interval $T = [t_s, t_e]$ and then update GK_T with $GK_T \leftarrow GK_T \cup \langle g,k,T \rangle$.

4.4 The XACML implementation

In this section we show how it is possible to use the XACML architecture described in Section 3 to implement, introducing only some minor changes, the RA-GBAC model of Section 4.3. The activation of a task is associated with a complex mitigation strategy, while, when asking for a specific permission, a subject always have to specify an active task associated with that permission. In order to obtain access, the task has to be assigned to a team of which the subject is a member. This lead us to the definition of two types of access request, both to be evaluated by the PDP: one for the activation of a specified task in behalf of a team and the other for obtaining a permission associated to a specified active task. We can distinguish between these two request-type using the attribute `action-id` of the `action` category that can assume a value in the set {`task-assignment`,`obtain-permission`}. Since a subject could be member of more than one team, when he requests to activate a task he has to specify in behalf of which team he is acting. Similarly, when requesting to access to a permission, the access subject has to specify, not only the team, but also the active task associated with the permission.

First of all we illustrate with an example how to write a complex risk mitigation `<PolicySet>` that can be associated with the activation of one or more tasks. Then we show how it is possible to express the rule for granting of a permission associated with an active task writing a single `<PolicySet>`. After that we discuss the introduction of some changes in the previously defined architecture to support the new features of our model.

4.4.1 A complex risk mitigation `<PolicySet>`

In Figure 4.2 we show an example of risk mitigation `<PolicySet>` with the same risk and budget thresholds of the risk mitigation strategy represented in Figure 3.8. We don't restrict the target of the `<PolicySet>` to any specific `task-id` value so we can refer to this risk mitigation strategy and associate it to more than one task. In Line 01 we specified the `first-applicable` rule-combining algorithm because it results in a more efficient evaluation of the policy. In Lines 02-04 we use the `<VariableDefinition>` element to specify the risk and budget thresholds. In Lines 05-12 we define a boolean variable called `team-task-check` that takes value `true` if and only if the access subject is a member of the specified team and the task has not been yet activated. We encode these two conditions in a variable because we need to check them in the `<Condition>` element of every `<Rule>` of the risk mitigation policy. Note that, in defining

[1] In practice, it is reasonable to "lock" a particular instance of a task only available to the requesting group, but different groups can be granted to perform different instances of a task.

this variable, we introduced, like we did for the `risk-value` attribute in Section 3, two new categories of attributes, one for the attributes of the team entity and the other for the attributes of the task entity. This is compliant with XACML 3.0 specification and allow us to easily distinguish between attributes of the access-subject and of its team, or between attributes of the permission and of the task to which the permission is associated. Like in our individual RAAC implementation we define a variable for every risk-range that check whether the risk-value for the access request is in that range. Note that, for the retrieval of the assessed risk value, we use the same attribute name as in our previous implementation but, in this case, the risk has to be assessed considering that is the whole team accessing to all the permissions of a task instead of an individual accessing to a single permission. Similarly to what we did in the XACML implementation of the individual RAAC model, we define, for every *i*-th budget threshold, a variable called `budget-check-`*i* that is evaluated to `true` if and only if the available budget for the team is greater than the threshold. To do so, we refer to the team budget as an attribute of the team (Line 23). At this point, the team budget could be calculated by the related PIP module as the sum of the budget of all the members. Alternatively we could require the different members to top-up in advance the team budget using their own individual budget. Recall that, to calculate the budget needed by a team to activate a task we use the Equation (4.2). The budget threshold is obtained by multiplying the budget associated with the range in which lies the risk value by the number of members of the team. In order to easily calculate this value for every rule, we define a numeric variable called `member-number` (Lines 13-16) that counts the members of the team associated with the request. The budget threshold is calculated applying the `double-multiply` XACML function to the previously defined `budget-`*i* and `member-number` variables (Lines 24-26).

The first `<Rule>` element (Lines 27-30) takes as applicability condition the boolean function `not` applied to the `team-task-check` variable and return the `Deny` decision, eventually together with some obligations. This rule means that, if the access-subject is not a member of the specified team or the task has been already activated, the access request has to be denied. The following `<Rule>`s express the decisions to be taken according to the risk mitigation strategy. For example the `<Rule>` with id `task:r:2` (Lines 33-39) returns the value `Permit` when the assessed risk-value is in the second range and the team-budget is greater than the calculated threshold. The condition element of this rule contains references to the variables `team-task-check`, `risk-check-1` and `budget-check-1` described above. In Line 46 we can express some system and collective obligations. We discuss later in this Chapter how to represent collective user obligations that apply to the whole team and system obligation for the deduction of the team budget using the XACML syntax for obligations.

Similarly the fourth `<Rule>` element (Lines 40-46) specifies that the value the request has to be denied when the risk-value attributes lies in the second range but the budget is below the calculated threshold. In this case, system obligations can be used to notify the reason why the request has been denied.

When an access request of the type `task-assignment` is evaluated to `Permit` the system has to assign the specified task to the requesting team. To implement this behaviour, we define a system obligation at the `<Policy>` level (Lines 52-59) setting the `FullfillOn` attribute to the `Permit` value. This means that this obligation will be returned when

the result of the `<Policy>` containing these rules evaluates to `Permit`. We assign the `system:assign:task` value to the `ObligationId` attribute and we encode the task and team ids in the `<AttributeAssignment>` elements.

```
00 <PolicySet PolicySetId="task:RMS:0" PolicyCombiningAlgId="permit-ovveride">
01  <Policy PolicyId="task:PO" RuleCombiningAlgId="first-applicable">
02   <VariableDefinition VariableId="risk-0">0.2</VariableDefinition>
03   <VariableDefinition VariableId="risk-1">0.7</VariableDefinition>
04   <VariableDefinition VariableId="budget-1">0.3</VariableDefinition>
05   <VariableDefinition VariableId="team-task-check">
06    <Apply FunctionId="function:and">
07     <Apply FunctionId="string-at-least-one-member-of">
08      <AttributeDesignator Category="subject" AttributeId="id" DataType="#string"/>
09      <AttributeDesignator Category="team" AttributeId="member_id" DataType="#string"/>
10     </Apply><Apply FunctionId="not">
11      <AttributeDesignator Category="task" AttributeId="active" DataType="#boolean"/>
12   </Apply></Apply></VariableDefinition>
13   <VariableDefinition VariableId="member-number">
14    <Apply FunctionId="integer-to-double"><Apply FunctionId="string-bag-size">
15      <AttributeDesignator Category="team" AttributeId="member_id" DataType="#string"/>
16   </Apply></Apply></VariableDefinition>
17   <VariableDefinition VariableId="risk-check-1">?(0<=risk<0.2)</VariableDefinition>
18   <VariableDefinition VariableId="risk-check-2">?(0.2<=risk<0.7)</VariableDefinition>
19   <VariableDefinition VariableId="risk-check-3">?(0.7<=risk<=1)</VariableDefinition>
20   <VariableDefinition VariableId="budget-check-1">
21    <Apply FunctionId="double-greater-than-or-equal">
22     <Apply FunctionId="double-one-and-only">
23      <AttributeDesignator Category="team" AttributeId="budget" DataType="double"/>
24     </Apply><Apply FunctionId="double-multiply">
25      <VariableReference Id="member-number"/><VariableReference Id="budget-1"/>
26   </Apply></Apply></VariableDefinition>
27   <Rule RuleId="task:r:0" Effect="Deny"><Condition><Apply FunctionId="function:not">
28    <VariableReference VariableId="team-task-check"/></Apply></Condition>
29    <ObligationExpressions>...</ObligationExpressions>
30   </Rule><Rule RuleId="task:r:1" Effect="Permit"><Condition>
31    <Apply FunctionId="function:and"><VariableReference VId="team-task-check"/>
32   <VariableReference VariableId="risk-check-0"/></Apply></Condition></Rule>
33   <Rule RuleId="task:r:2" Effect="Permit">
34    <Condition><Apply FunctionId="function:and">
35     <VariableReference VariableId="team-task-check"/>
36     <VariableReference VariableId="risk-check-0"/>
37     <VariableReference VariableId="budget-check-1"/>
38    </Apply></Condition>
39   <ObligationExpressions>...</ObligationExpressions></Rule>
40   <Rule RuleId="task:r:3" Effect="Deny">
41    <Condition><Apply FunctionId="function:and">
42     <VariableReference VariableId="team-task-check"/>
43     <VariableReference VariableId="risk-check-1"/>
44     <Apply FunctionId="function:not"><VariableReference VariableId="budget-check-1"/>
45    </Apply></Apply></Condition>
46   <ObligationExpressions>...</ObligationExpressions></Rule>
47   <Rule RuleId="task:r:4" Effect="Permit"><Condition><Apply FunctionId="function:and">
48     <VariableReference VariableId="team-task-check"/>
49     <VariableReference VariableId="risk-check-2"/></Apply></Condition>
50   <ObligationExpressions>...</ObligationExpressions></Rule>
51   <ObligationExpressions>
52    <ObligationExpression ObligationId="system:assign:task" FulfillOn="Permit">
53     <AttributeAssignmentExpression AttributeId="team_id">
54      <AttributeDesignator Category="team" AttributeId="id" DataType="#string"/>
55     </AttributeAssignmentExpression>
56     <AttributeAssignmentExpression AttributeId="task_id">
57      <AttributeDesignator Category="task" AttributeId="id" DataType="#string"/>
58     </AttributeAssignmentExpression>
59 </ObligationExpression></ObligationExpressions></Policy></PolicySet>
```

Figure 4.2: Risk mitigation policy for task assignment

4.4.2 The assigned-task `<PolicySet>`

Recall that, according to our RA-GBAC model, in order to get access to a set of permissions, a team has to activate the relative task. When a task has been activated and assigned to a specific team, all the permissions associated with the task will be granted to all the members of the team during the time-interval in which the task remains active. When requesting for a permission, the subject is required to specify to the context-handler in behalf of which team he is acting and for the fulfilment of which task he needs to access to the permission. The context-handler have to include these information in the request context.

In Figure 4.3 we show a `<PolicySet>` in which we express the rule that allows a subject, acting in behalf of a team, to access to the permissions associated to a task activated by the specified team. The `<Target>` element (Lines 01-04) restricts the applicability of the `<PolicySet>` only to the requests with the `obtain-permission` value assigned to the `action-id` attribute of the `action` category. It contains only one `<Policy>` element (Lines 5-45) with one inner `<Rule>` element (Lines 07-45). Since in our model all the risk mitigation methods are applied when a task is activated, when its condition is met, this `<Rule>` returns a `Permit` response without any user or system obligation. The `<Condition>` element (Lines 8-44) applies a logic `and` function to a sequence of boolean functions. The first `<Apply>` element (Lines 09-12) checks whether the access subject is member of the team specified in the request context. From Lines 13 to 26 we check whether the specified task has been activated and assigned to the team. The `<Apply>` element from line 27 to 38 returns `true` if the task is not yet expired and the one from line 39 to 44 check returns `true` if the requested permission is in the permission-set of the specified task.

The `<PolicySet>` doesn't refer to any specific subject, team, task or permission thus it can be used as a generic top-level policy for any requests of the type `obtain-permission`. We need to include only one policy like this in our PAP to manage all the possible requests for a specific permission. The policy combining algorithm for the `<PolicySet>` element is set to `deny-unless-permit`. This means that, if the condition of the only `<Rule>` in the policy is evaluated to `false`, the `<PolicySet>` will return a `Deny` response.

4.5 Representing and enforcing collective obligations

Similarly to what we have done for our individual RAAC implementation, when we write a RA-GBAC risk mitigation policy, we have to include, in every `Permit <Rule>`, an obligation for the deduction of the correspondent team budget value. In Figure 4.4 we show an example for a collective deduct-budget obligation that could be included in Line 46 of our example RA-GBAC risk mitigation policy (Figure 4.2) Instead of specifying the access-subject we specify the access-team from which the deposit has to be deducted (Lines 01-03). In Lines 05-09 we calculate the deposit amount that have to be deducted. Depending on how we choose to share the deposit between the team members, the amount could be deducted directly from the team-budget or from the member's budget values.

When specifying collective user obligations we can use the same syntax and semantic of the individual user obligations defined in Section 3.2. Instead of specifying the user, we have to specify the team that have to fulfil the obligation. The main difference is in the implementation of

```
00 <PolicySet PolicySetId="assigned:task:PS" PolicyCombiningAlgId="deny-unless-permit">
01  <Target><AnyOf><AllOf><Match MatchId="string-equal">
02   <AttributeValue DataType="#string">obtain-permission</AttributeValue>
03   <AttributeDesignator Category="action" AttributeId="action-id" DataType="#string" />
04  </Match></AllOf></AnyOf></Target>
05  <Policy PolicyId="assigned:task:PO" RuleCombiningAlgId="permit-overrides">
06   <Target></Target>
07   <Rule RuleId="assigned:task:R" Effect="Permit">
08    <Condition><Apply FunctionId="function:and">
09     <Apply FunctionId="string-at-least-one-member-of">
10      <AttributeDesignator Category="subject" AttributeId="id" DataType="string"/>
11      <AttributeDesignator Category="team" AttributeId="member_id" DataType="string"/>
12     </Apply>
13     <Apply FunctionId="string-at-least-one-member-of">
14      <AttributeDesignator Category="team" AttributeId="id" DataType="#string"/>
15      <AttributeDesignator Category="task" AttributeId="team_id" DataType="#string"/>
16     </Apply>
17     <AttributeDesignator Category="task" AttributeId="activated" DataType="#boolean"/>
18     <Apply FunctionId="dateTime-greater-than-or-equal">
19      <Apply FunctionId="dateTime-one-and-only">
20       <AttributeDesignator Category="environment" AttributeId="current-dateTime"
21         DataType="dateTime"/>
22      </Apply>
23      <Apply FunctionId="dateTime-one-and-only">
24       <AttributeDesignator Category="task" AttributeId="start_time"
25        DataType="dateTime"/>
26     </Apply></Apply>
27     <Apply FunctionId="dateTime-greater-than-or-equal">
28      <Apply FunctionId="dateTime-add-dayTimeDuration">
29       <Apply FunctionId="dateTime-one-and-only">
30        <AttributeDesignator Category="task" AttributeId="start_time"
31          DataType="dateTime"/>
32       </Apply><Apply FunctionId="dayTimeDuration-one-and-only">
33        <AttributeDesignator Category="task" AttributeId="duration"
34         DataType="#dayTimeDuration"/>
35      </Apply></Apply>
36      <Apply FunctionId="dateTime-one-and-only">
37       <AttributeDesignator DataType="dateTime" AttributeId="current-dateTime" />
38     </Apply></Apply>
39     <Apply FunctionId="string-at-least-one-member-of">
40      <AttributeDesignator Category="task" AttributeId="permission_id"
41        DataType="#string"/>
42      <AttributeDesignator Category="permission" AttributeId="permission_id"
43        DataType="#string"/>
44    </Apply></Apply></Condition>
45 </Rule></Policy></PolicySet>
```

Figure 4.3: Assigned task policy

```
00    <ObligationExpression ObligationId="system:deduct-budget">
01     <AttributeAssignmentExpression AttributeId="access-team">
02      <AttributeDesignator AttributeId="team-id" Category="team"/>
03     </AttributeAssignmentExpression>
04     <AttributeAssignmentExpression AttributeId="budget-value">
05      <Apply FunctionId="double-multiply">
06       <VariableReference VariableId="member-number"/>
07       <VariableReference VariableId="budget-1"/>
08      </Apply>
09     </AttributeAssignmentExpression>
10    </ObligationExpression>
```

Figure 4.4: Collective deduct-budget `<ObligationExpression>`

the obligation monitor that will have to recognize, for every user action, if the action fulfils any of the obligations assigned to any of the teams of which the user is a member.

4.6 Extended architecture

In this section we describe how we can extend the architecture proposed in Section 3.4 to support our XACML RA-GBAC implementation. In the `<PolicySet>` of Figure 4.2 we refer, using the `<AttributeDesignator>` element, to attribute of the categories `team`, `task` and `permission`. These attribute categories are not supported by the standard XACML architecture an we need to implement, like we did for our `risk-assessment-module`, a PIP module that support the retrieval of those categories of attributes. The module that supports the team category in particular has to provide a method for the retrieval of the available team-budget value. This method can be implemented in different ways depending on how we define the sharing of the deposit between members of the same team: it could just return the sum of all the deposit budget of the single members or we could decide to maintain in our data-source a team-budget value that can be topped up by the members using their individual budgets. Moreover the risk assessment module needs to be changed in order to account the risk derived by granting the whole set of permissions to all the members of the requesting team. In our model we don't specify how the risk value should be calculated. It is important to recall that the risk-assessment module of the PIP can access to all the attributes in the request-context and can interact with the other PIP modules to retrieve all the attributes that it could need for evaluating the risk associated to the request.

The modular architecture for the handling of system obligation doesn't need to be changed. In fact we just need to implement a module for the fulfillment of the `system:deduct-budget` obligation according to the budget-sharing mechanism that we have chosen.

The modular architecture of the user-obligation monitor defined for our individual RAAC model remains applicable for the development of the risk-aware GBAC model implementation. When a new collective obligation arise, we record to which team every obligation is associated. Like in modular RAAC obligation monitor implementation, for every collective obligation there has to be a module that implements its monitoring. For every action performed by a subject, the modules have to check whether the action fulfils one of the obligations that are in the pending obligations list and are associated to one of the team of which the subject is a member.

4.7 Discussion

In this section we discuss some related work on the field of group-based access control and the representation and enforcement of collective obligations.

Recently, interest has been growing in the field of team-based access control as an approach to applying access control in collaborative environments such as those involving work-flows. Kalam et al. (2003) proposed an organization-based access control (OrBAC) model in which the concepts of role, activity, and view are defined to groups (set of users), actions and objects respectively. This model provides the basis for their later comprehensive study of obligation policies. In recent related works, Elrakaiby et al. (2009, 2012) and Cuppens et al. (2013) introduced the notion of group contexts to represent group relationships and to allow the policy designer to define shared responsibilities and alternative duties between more subjects. The limit of this model is that it does not give us the capability to express every kind of team obligation, in fact responsibilities can only be shared between subjects empowering the same role, and in particular between all (universally quantified) or only one of the subjects related to that role.

Thomas (1997) explored the possibility of introducing the notion of team in an RBAC system to provide an access control scheme for collaborative environments. Two main requirements have been identified by Thomas:

1. the need for a hybrid access control model that:
   (a) incorporate the advantages of role-based permission across object types; and
   (b) allow to specify fine-grained, identity-based control on individual users in certain roles, and to individual object instances.
2. the need to distinguish the passive concept of permission assignment from the active concept of context-based permission activation.

These two requirements are derived by the study of an healthcare scenario with clinical staff collaborating in a team. Teams are dynamically formed as collections of staff members, each adopt one or more roles. At any moment a member can join or leave the team. Teams can be assigned to different patients. Data from a patient can be accessed only by members of the team to which he has been assigned. The permission that every member has to access clinical records should reflect his/her role in providing care to a specific patient. These permissions should be deactivated when the patient is discharged. Thus we want to be able to assign permissions to a particular user/team regarding a specific object and not all the objects of the same type. As Thomas states in his work, the basic RBAC scheme cannot be used to enforce the requirements 1.(a) and 1.(b) together. Trying to satisfy both of these, he proposed TeaM based Access Control (TMAC), where team definitions include the collaboration context that is needed to satisfy the requirement 1.(b). Every team is associated with certain resources, and users who belong to a team, are assigned a subset of a set of Team roles TR and given access to resources available to the team. However, the effective permissions of a user are always derived from permission types defined for roles that the user belongs to. The main limitation of this model is that it restricts the roles that a user can take on while joining a team. The ideas behind this model have major influence on the development of our risk-aware GBAC model.

In related works Royakkers and Dignum (2000) and Grossi et al. (2004) investigate the representation, formalization and enforcement of collective obligations. They introduce concepts as commitment, delegation and collective plans. A collective obligation is represented as a complex task, composed of different subtasks. They propose an approach based on the analysis of the collective plan, and of the fulfilment of single sub-tasks to understand which users have to be considered responsible for the failure in the fulfilment of a collective obligation. We believe that, for our purpose there is no need to represent collective obligation as complex tasks. In access control, in fact, obligations are usually simple tasks that can be performed by one of the members. Instead of applying such an analysis to understand who we have is responsible for a failure, we propose an approach based on the budget deposit that allow the users to decide how much they can be considered responsible for the fulfilment of an obligation.

Another approach is taken by Garion and Cholvy (2007) that propose a way to derive a set of individual obligations from a collective obligation. In case of failure in the fulfilment of an obligation, they state that the whole group have to be considered responsible. This approach is

similar to ours but we provide a way for the members to share responsibilities in a more flexible way so that every user can decide how much he can be considered responsible for the fulfilment of an obligation.

4.8 Summary

In this chapter we have defined a formal abstract model for implementing a RA-GBAC system. We then proposed an XACML improved architecture showing how it is possible to write a set of standard XACML policies that implement our model. We propose a modular architecture that lets the system administrators free to implement custom risk-assessment function and budget sharing mechanism. These two components can be implemented respectively in the application dependent risk-assessment module of the PIP and in the obligations handler of the PEP. However we haven't proposed any mechanism or strategy for the definition of these two modules. In the next chapter we propose a more detailed scenario that brings together elements from the two previous running examples. While discussing the scenario and showing how our model applies to it, we propose some risk-assessment and budget sharing mechanism and underline some issues about the model and its implementation in a real system.

Chapter 5

Healthcare Scenario and Discussion

In this chapter we present a more detailed example based on the healthcare scenario introduced in Section 4.1, showing why the RAAC model described in Section 2.1.1 is not flexible enough when handling risky permissions in a collaborative environment and thus motivating the introduction of our risk-aware GBAC model. We use this example to discuss issues and alternative approaches to develop RA-GBAC models. We show in detail, providing some example risk and budget values, how the model and its XACML implementation work. While discussing the example, we make some remarks on how to write a coherent risk mitigation strategy for a task that contains multiple permissions, given the risk mitigation strategies for those single permissions. We discuss in detail some possible choices for the team risk-assessment algorithm and the budget-share mechanism. We also consider the possibility to defining some additional security constraints at the model.

5.1 Healthcare example scenario

After a car accident, Alice is taken by an ambulance to the nearest hospital. The hospital is facing an emergency situation and most of the clinical staff members are already busy in other emergency interventions. Bob is a *surgeon* who is working in the emergency room of the hospital. Carol is a *nurse trainee* and at the moment is the only nurse that is not busy in other interventions. Charlie is an *anaesthetist trainee*, and is the only anaesthetist not busy in other interventions. The relevant part of the role hierarchy of this hospital is showed in Figure 5.1.

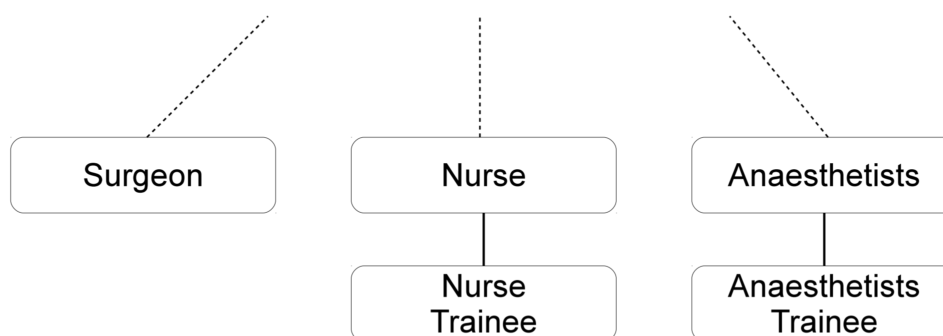

Figure 5.1: Scenario role hierarchy

A *nurse* can access to all of the permission associated with the *nurse trainee* role while a *nurse trainee* can access only to some of the *nurse* permissions. An *anaesthetist* can access to

all of the permission associated with the *anaesthetist trainee* while an *anaesthetist trainee* can access to only some of the permission of an *anaesthetist*. Every employee of the hospital act empowering one or more roles and can temporally activate other roles according to the risk-aware RBAC model. The budget values for every user are showed in Table 5.1.

Bob	0.95
Carol	0.25
Charlie	0.3

Table 5.1: Individual budget values

After a quick visit, Carol determines that Alice needs to be brought to the emergency room for a surgical operation. To operate, Bob needs the assistance of a nurse and an anaesthetist but there are not any one available other than, respectively, Carol and Charlie. Carol and Charlie would like to temporally activate respectively the *nurse* and the *anaesthetist* roles. Let's consider the case in which the Carol and Charlie try to activate the respectively needed roles according to the previously defined risk-aware RBAC model. Recall that the activation of a role to which the user is not normally allowed is associated with a risk mitigation strategy. The risk mitigation strategies for the activation of the two roles are showed in Table 5.2 and Table 5.3

Nurse RMS			
Risk	0.2	0.5	0.8
Budget	0	0.3	0.4
Obligations	$\perp$	O_1	O_2

Table 5.2: Nurse RMS

Anaesthetist RMS			
Risk	0.2	0.45	0.75
Budget	0	0.35	0.4
Obligations	$\perp$	O_3	O_4

Table 5.3: Anaesthetist RMS

Every column represents a risk range. For $0 \leq i \leq n$, let r_i be the risk thresholds of the risk mitigation strategy. We define constant values for r_0 as 0 and for r_n as 1. For $1 \leq i \leq n-1$, r_i takes the value on the first row of the *i*-th column. The second and third rows of the *i*-th column define the needed budget value and the set of obligations to be returned when the assessed risk value is between r_{i-1} and r_i. When the risk value is between r_{n-1} and r_n the request is always denied. For example the second column of Table 5.2 specify that, when $0.2 <$ risk-value ≤ 0.5 the needed budget value is 0.3 and the obligations in O_1 have to be returned.

The risk-assessment algorithm used in the scenario system assess a risk-value of 0.47 for Carol activating the *nurse* role. This means that Carol would need to use 0.3 budget for accessing to the permissions associated with the nurse role. As we can see in Table 5.1 Carol has only 0.25 available budget value so is not able to activate the *nurse* role. In the same way, the system assessed a risk-value of 0.35 for Charlie activating the *anaesthetist* role thus, according to Table 5.3, Charlie would need to use a deposit of 0.35 to obtain the permission but his budget-value is

only 0.3.

It could be reasonable in this situation for Bob, Carol and Charlie to form a team in order to share the responsibilities for the risk incurred by allowing Carol to activate the nurse role and Charlie to activate the anaesthetist role. In particular Carol and Charlie could use some of Bob's budget to temporally activate their roles. This would be an advantage for Carol and Charlie, that could access to some permission using part of Bob's budget. Bob would be partly responsible for both Carol and Charlie, and could lose part of his budget if they misuse their permission or if nobody fulfil some of the resulting obligations but, doing so, he could proceed with the surgical operation.

5.1.1 Task definition and its RMS

Recall that, in our RB-RAAC model, a user can empower one or more roles and can submit a request to the system in order to temporarily activate a role to which he is not normally allowed. Every role activation is seen as a single permission and is associated with a risk mitigation strategy. We now want to apply our RA-GBAC model into the example scenario. Recall that, to fulfil their job, Bob, Carol and Charlie need the permissions to activate all the roles `surgeon`, `nurse` and `anaesthetist`. According to the RA-GBAC model, we need to create a task that groups together all these permission for a time-interval large enough to allow them to complete the surgical intervention. We named this task as `emergency:intervention:1`. Note that, in this particular case, since we know in advance that, in the requesting team, there is a staff member that is already allowed to empower the surgeon role and he is the only one who need to use those permissions, it would not be necessary to include this permission in the task's permission-set. In a general case, however, when defining a task, we don't know which team is going to request to fulfil the task, thus, we have to include all the permissions that are necessary for the task to be fulfilled by a generic team. The fact that one of the members of the requesting team is already enabled to activate the surgeon role should reflect in a lower risk-value returned by the risk-assessment algorithm.

This task has then to be associated with a risk mitigation strategy. Our risk-aware GBAC model has to be defined considering the possible consequences of granting to all the members of a team all the permissions in the task's permission-set for the time-interval needed to complete the task. Recall that the needed-budget evaluation (Equation 4.2) and the risk-assessment algorithm already take into account the size and the composition of the group's members set. The risk mitigation strategy for the task has to be defined considering only which permissions and for how long they have to be granted. According to our model, others factors based on the context of the system (e.g. the task criticality) have to be considered in the risk assessment procedure. In some scenarios, the risk mitigation strategies for some predefined complex tasks could be provided a priori by the system administrator. However in other cases it's not possible to know in advance all the possible types of task that could need to be fulfilled in a system. For this reason, it would be interesting to define a mechanism that, given a task, its duration and the risk-mitigation strategies for the permissions required to fulfil it[1], calculates and returns a RMS to associate with it. This is outside the scope of our work and, in this example, we will use a predefined RMS for the

[1] Recall that in the RA-GBAC model there is no definition of RMS for a single permission, it would be useful to maintain this definition in case we want to automatically calculate the tasks' RMS when needed.

`emergency:intervention:1` task (Table 5.4).

Task RMS			
Risk	0.2	0.5	0.8
Budget	0	0.3	0.45
Obligations	$\perp$	O_5	O_6

Table 5.4: Task RMS

5.1.2 Team risk calculation

When a user wants to access to some permission, in behalf of a team, in order to fulfil a task, firstly he has to send a request to the access control system to be assigned to the specific task. The task is associated to a risk mitigation strategy and, in order to take a decision, the system has to assess the risk incurred by granting the permission-set associated with the task to all the members of the team. We propose an approach for the risk assessment procedure that consists in two steps:

1. Firstly, for every user we have to assess the risk incurred by granting to him the set of permissions associated with the task for the duration of the task.
2. After that we have to calculate, given the risk associated to every single user, the combined risk for the whole team

In our abstract model we don't define any specific mechanism for the risk assessment for an individual. Moreover, in our XACML based risk-aware GBAC implementation, we provide an interface to implement a risk assessment module. The developer of an access control system based on our model is free to define his own risk assessment algorithm and to implement it into the risk-assessment module. The lack of a risk-assessment algorithm for individual user-permission assignment is certainly ont of the issues that should be addressed by future works. We now suppose that we already have assessed a risk-value for every user-permission assignment and, given these values as input, we propose a strategy for the assessment of the risk associated with a team-task assignment.

Let K be a task, t be the time-interval in which K remains active and $P = \{p_1, p_2, \ldots, p_n\}$ be the set of permissions associated with K. Let $G = \{u_1, u_2, \ldots, u_m\}$ be a group of m users who requests to fulfil K. We call $r_{(i,j)}$ the risk assessed by the system for granting to the i_{th} member of G the j_{th} permission of K. Recall that the risk associated with a subject-permission assignment reflects the likelihood of the permission being misused by that subject. The risk incurred by assigning a task to an individual subject should represent the likelihood of that subject misusing at least one of the permissions included in the task's permissions set during the time-interval in which the task remains active. We take a probability calculation approach and we propose a formula for the calculation of r_i as the risk incurred by granting the permission to fulfil K to the user u_i.

$$r_i = 1 - \prod_{j=1}^{n} (1 - r_{i,j}) \tag{5.1}$$

where $r_{i,j}$ is the probability of u_i misusing the permission p_j. Thus $1 - r_{i,j}$ represents the probability that u_i doesn't misuse permission p_j and the product is the probability of the event "u_i doesn't

misuse any of the permissions of the task". Our approach is only one possibility and it doesn't take into account a lot of factors like the criticality of the task, any kind of correlation between the different permissions or the duration of the task. However we believe that in most of the cases it could make sense to calculate the risk in this way.

Considering our scenario we suppose that, using the mechanism described in Formula 5.1, we assessed, for the task `emergency:intervention:1`, the r_i values showed in Table 5.5.

User	r_i
Bob	0.1
Carol	0.5
Charlie	0.5

Table 5.5: Individual task-user assignment assessed risk

Once we have assessed a value r_i for every member of the group u_i we can combine these values in different ways to obtain the risk r for the assignment of K to the team G. In Formula 4.3 we propose an approach that consists on taking the maximum value between all the r_i. This is a really simple mechanism that is based on the assumption that the less trustworthy member is the one who is more likely to misuse his permission. However this mechanism doesn't take into account the risk of other members misusing one of the permissions of the task's permission-set. A more complex risk calculation algorithm that considers the cumulative risk of all the users misusing their permission could be based on a probability computation. With an approach similar to the one taken when computing r_i (Formula 5.1) we calculate r.

We want to calculate the probability of at least one user u_i misusing one or more of the task's permissions. We call this event E. For every member of the team u_i, $1 - r_i$ represents the probability of him not misusing any of the permissions. The product $\prod_{i=1}^{m}(1 - r_i)$ represents the probability of all the users not misusing any of their permissions, that corresponds to the event $\neg E$. We can thus calculate the probability of E as $P(E) = 1 - P(\neg E)$ that lead us to the Formula 5.2.

$$r = 1 - \prod_{i=1}^{m}(1 - r_i) \tag{5.2}$$

Although we believe that Formula 5.2 provides a more appropriate risk calculation method than the one based on the maximum individual risk, it doesn't take into account some additional factors like the mutual control that every member exercises over the others when acting together and sharing responsibilities. In fact we believe that the actual risk-value associated with a team-task assignment should be considered slightly lower than the value estimated with the probability computation of Formula 5.2.

However, in our scenario implementation we use a risk-assessment PIP module that implements this mechanism. Using the already assessed risk-values for the individual-task assignment presented in Table 5.5 we assess a risk-value for G fulfilling the task K of $r = 0.775$. According to the risk mitigation strategy of Table 5.4 and the `Bud` function defined in Formula 4.2, the system, before assigning K to G, has to check whether the team has an available budget equal or greater than $B = b_i * |G| = 0.45 * 3 = 1.35$.

5.1.3 Budget sharing mechanisms

Recall that the way in which the access control system check for the availability of the team-budget depends on how the budget-sharing mechanism is implemented. In Section 4.3 we proposed two different approaches:

1. Defining, in the data-source / PIP, a team-budget variable initialized to 0. Every user, at any time, can top-up the team-budget using his own budget.

2. Defining the team-budget as the sum of all the budget-values of the team-members. The system decides how much of his budget every users has to use in order to get access.

We now discuss in detail how these two approaches can be implemented, outlining advantages and disadvantages of both the solutions. Recall that in the XACML risk mitigation strategy for the activation of a task (Figure 4.2) we refer to the team-budget as an attribute of the `team` category.

In the first case the PIP module that supports the retrieval of that attribute has just to return to the PDP the value of the team-budget attribute. This value has to be previously topped-up by the team members using their own individual budget. The needed budget value is deducted from the team-budget when the permission for activating a task is granted to a team and is returned only if all the obligations have been fulfilled. In case we want to be possible, for the members to ask for having back their budget, we would need to define a mechanism that allow the system to decide how much budget every user can get back. Recall that part of the team-budget could have been lost by the team as a punishment for not having fulfilled some obligations. We propose a mechanism in which, every user can leave the team and take back a part of the team-budget that is proportional to the value he has topped-up compared to the ones from the other users. It is reasonable to allow a user to do so only in a situation in which there are no active tasks and pending obligations for the related team. For every member $u_{(i,j)}$ with $0 < i < n$ of the team G_j we maintain a variable $b_{(i,j)}$ that represents his contribution to the actual team-budget value. Let B_j be the amount of budget currently available for the team. At any moment:

$$\sum_{i=0}^{n-1} b_{(i,j)} = B_j \tag{5.3}$$

When a user u_i decides to top-up the team-budget B_j with an amount v taken from his individual budget, the variables $b_{(i,j)}$ and B_j have to be updated according to the following formulas:

$$b_{(i,j)} = b_{(i,j)} + v$$

$$B_j = B_j + v$$

When a team obtains the permission to activate a task using a budget value w, every value $b_{(i,j)}$ has to be updated according to the following formula:

$$b_{(i,j)} = b_{(i,j)} * \frac{(B_j - w)}{B_j} \tag{5.4}$$

After that the amount w has to be deducted from the team budget B_j. If all the obligations returned by the PDP have been fulfilled by the team, all the deducted values, from B_j and $b_{(i,j)}$, have to

be given back. At any time a user can decide to leave the team and take back his budget-value $b_{(i,j)}$. The idea behind this mechanism is that, when we deduct a value w from the team budget B_j, we want to scale every value $b_{(i,j)}$ in a way that respects the proportion between all these values that the users decided to put and that maintains the validity of Equation 5.3. Let's consider, for example, the values of Table 5.1. Bob, Carol and Charlie join a team G_1 and decide to put respectively 0.5, 0.2 and 0.3 for a total team budget value B_1 of 1.0. Bob decide then to activate a task that requires to use 0.4 of the team budget. B_1 takes the value 0.6. The value $b_{(i,1)}$ for Bob, Carol and Charlie are updated according to the Equation 5.4.

$$b_{(\text{Bob},1)} = \frac{0.5 * 0.6}{1.0} = 0.3$$

$$b_{(\text{Carol},1)} = \frac{0.2 * 0.6}{1.0} = 0.12$$

$$b_{(\text{Charlie},1)} = \frac{0.3 * 0.6}{1.0} = 0.18$$

Note that the Equation 5.3 remains valid, in fact:

$$b_{(\text{Bob},1)} b_{(\text{Carol},1)} + b_{(\text{Charlie},1)} = 0.18 + 0.12 + 0.3 = B_1 = 0.6$$

This first approach is a safer solution for a user with a high budget that has agreed to form a team with some other user with lower budget. In fact every member is free to decide how much budget to put in order to activate a task, and thus how much he is to be considered responsible in case of the permission being misused or one of the obligations not being fulfilled.

In the second case, when the PIP module that supports the attributes of the `team` category is asked to return the team-budget value, it has to retrieve all the budget values of the team's members and to return their sum to the PDP. If the budget is equal or higher than the needed value, the task will be assigned to the team and the access control system has to decide how much deposit to deduct from every user's individual budget. This approach is transparent to the users that don't need to manage their budgets and to know how the system works in detail. On the other hand it is more risky for a user with a high budget value to join a team with some untrustworthy users. In fact, since he has no control in how much of his budget will be used for the activation of a task, the failure in fulfilling one of the obligations could result in the user loosing a large part of his budget. We now describe in detail a mechanism that implements an equal sharing of the budget showing how it applies to the example scenario and then mention some alternative approaches to decide how much deposit to take from every team member. Let B_K the budget needed for the activation of the task K and $|U|$ be the number of members of the team G. Ideally we want every member to put an amount of budget equal to $\frac{B_K}{|U|}$. If one of the members doesn't have enough budget, the other users will have to equally contribute an additional deposit to cover the missing value. In Algorithm 1 we show how we can calculate the amount of required budget for every user according to this principle In every loop we check whether every user has an amount of budget higher than the one needed to equally share the remaining deposit. If not, we deduct from every user an amount equal to the minimum between all the member's budget. In the following loops we try to equally divide the remaining deposit between the members that still have some available

Algorithm 1 Equal budget share algorithm

```
B_K ⇐ needed budget
U = {u_i : 1 ≤ i ≤ n} ⇐ team members
∀i : 1 ≤ i ≤ n | b_i ⇐ budget available for user u_i
while B_K > 0 do
  e ⇐ B_K/n
  b_min ⇐ min{b_i : 1 ≤ i ≤ n}
  v ⇐ min{e, b_min}
  for i = 1 to n do
    b_i ⇐ b_i − v
  end for
  B_K ⇐ B_K − v ∗ n
  for i = 1 to n do
    if b_i = 0 then
      U ← U \ u_i
    end if
  end for
  n = |U|
end while
```

budget.

Let's consider the activation of the task `emergency:intervention:1` by team G. Recall that the needed budget for its activation is 1.35, that the available budget values for the members of G are shown in Table 5.1 and that its sum evaluates to 1.5. The access control system grants to G the permission to activate the task and has to decide how much budget every user is going to pay. In the first loop the system take 0.25 from all the members of the team and remove Carol from set of user's with any available budget. Bob and Charlie remains respectively with 0.70 and 0.05 of budget. B_K takes the value $1.35 - 0.25 * 3 = 0.60$. In the following loops the system deducts 0.05 from both Bob and Charlie's budget and then deducts the remaining 0.50 from Bob's budget. Charlie and Carol have no budget left while Bob remains with 0.15 budget. Another approach could consists on using, for every member, a budget value proportional to the risk assessed for the single user activating the task (or, if defined, to his trustworthiness).

5.1.4 Security thresholds and constraints

We now discuss the possibility of defining some additional security constraints in the RA-GBAC model. According to the mechanisms defined in Section 5.1.3, it is possible, for a user with low budget, to join a team and get access to all the permission activated for that team. This user cannot contribute to the team budget with a significant deposit but will get all the benefits of the other members. To avoid the occurrence of these situations it would be reasonable to define a threshold for a minimum budget contribution to join a team or to be granted the permissions associated with a task. One possible approach could consist in defining, in every risk mitigation strategy, a minimum value of budget that every user has to contribute. This value could be fixed or expressed as a fraction of the total required budget. Moreover, when granting access to a team for the activation of a particular task, we could require to avoid granting access to untrustworthy subject for which the risk, calculated for the subject acting as an individual, lies in the last risk range. One solution could consist in calculating all the risk associated with the members of a team

acting as individuals and grant access to the team only if the maximum of these values is lower than the last threshold.

A limit of our work consists in the fact that none of our models supports the specification and enforcement of *separation of duty* constraints. Moreover, it would be interesting in some situation to be able to determine which of the team members have to be considered responsible for the failure in the fulfilment of a collective obligation. This would involve consideration about the team plan for the fulfilment of the obligation and about the capability of the different user to fulfil the different sub-tasks of which the obligation can be composed. This is outside the scope of our work and we take a different approach, allowing the user to share responsibility choosing how much of their budget to deposit.

5.1.5 Summary

In this chapter we presented a scenario example that brings together elements from the two examples in Section 2.1.1 and 4.1 and discussed some issues about our abstract model and how it can be implemented in a real scenario. In particular we proposed some mechanisms for risk-assessment and budget sharing and defined more in detail some algorithms for their implementations. We suggested a strategy for writing a coherent risk mitigation strategy to be associated with a task that groups together different permissions and being assigned to a generic team. We outlined which elements of risk should be considered by the risk-assessment module and which ones we have to take into account during the definition of the risk-mitigation strategy.

Chapter 6

Conclusion

In this chapter we present a summary of the main contributions of this thesis.

The goal of our work was to study the development of novel access control models that address the needs to securely share informations in a dynamic collaborative environment, such the ones in which different subjects can group together, joining a team to collaborate at the fulfilment of a common goal. We believe that RAAC models can successfully manage these kind of situations through an appropriate risk-assessment that takes into account the context informations, the team size and composition, the criticality of the permissions, and which is the goal or the task that the team is willing to fulfil by using the requested permissions. For these reasons, the definition of an appropriate risk-assessment mechanism is often application dependent. We thus decided to develop our model in a modular way so that, for every implementation of this model in a real scenario, the system administrator can define his own risk-assessment algorithm. The main contributions of this work are summarized as follows:

- We developed an abstract risk-aware access control model. Our work is based on the model proposed by Chen et al. (2012) and it supports dynamical risk assessment, risk authorization decision making and system obligation as risk mitigation methods. It uses risk mitigation strategy to decide, according to the range in which the assessed risk-value lies, whether to grant or not access for the requesting subject to a permission. We then extended this model introducing some minor changes to support user obligations and the concept of budget as a way to encourage the subjects to fulfil their obligations.

- We extended the RAAC model and adapted it for the management of permissions in a collaborative environment. We introduce the concept of task as a set of permissions associated with a validity time-interval and the concept of team as a dynamic set of users that decide to group together and collaborate in order to fulfil a common goal. At every team we associate a team-budget value that uses the individual budget of the members. According to this value, and to the risk mitigation strategies associated with the tasks, team can request the permissions to fulfil a task. Risk mitigation strategy can return collective obligations as risk mitigation methods. These are obligations that must be fulfilled by the team members before the expiration of a given deadline.

- We proposed an extended architecture, based on the reference XACML architecture, for implementing our models in an XACML-based system. This architecture supports the definition of risk mitigation strategies for tasks and permissions using standard features of the

XACML policy language. Dynamical risk assessment is seen as a normal attribute retrieval. In fact we define a module for the PIP that assess and return the risk-value associated with a request. This solution does not conform with the PIP definition. In fact the PIP modules should only retrieve values from the data-source without processing them. However this solution allow us to define a generic risk-assessment algorithm that is not limited by the feature of XACML language. We also implemented in the enforcement point an obligation handler that perform system obligations and monitor the fulfilment of user and collective obligations.

- We proposed a semantic for user, collective and system obligations definition based on the XACML syntax for the system obligation definition. The three kind of obligations are distinguished using a prefix on the `obligation-id` attribute. For a user or collective obligation we defined special attributes for the user or the team that is responsible for its fulfilment and for the specification of the deadline.

We believe that, with this work, we have developed a flexible model for the implementation of a generic RAAC system that can be easily built upon an existing access control system (e.g. based on RBAC). The model supports *user obligations* and *system obligations* and can be easily extended to support new types of mitigation methods. The introduction of the budget deposit mechanism encourages user to fulfil their obligation. We also introduced a mechanism that allows sharing of permissions and responsibilities through. Moreover, the proposed implementation is compliant with the XACML standard and thus supports all its features, like the development of distributed access control system, policy hierarchy and separation of decision and enforcement point.

Bibliography

American National Standards Institute (2004). *American National Standard for Information Technology – Role Based Access Control*. American National Standards Institute. ANSI INCITS 359-2004.

Baracaldo, N. and Joshi, J. (2012). A trust-and-risk aware rbac framework: tackling insider threat. In *Proceedings of the 17th ACM symposium on Access Control Models and Technologies*, SACMAT '12, pages 167–176.

Bell, D. E. and LaPadula, L. J. (1975). Secure computer systems: Unified exposition and Multics interpretation. Technical Report MTR-2997, The MITRE Corp.

Bijon, K. Z., Krishnan, R., and Sandhu, R. S. (2012). Risk-aware RBAC sessions. In *Proceedings of the 8th International Conference on Information Systems Security*.

Bishop, M. (2003). *Computer Security: Art and Science*. Addison-Wesley.

Chen, C., Han, W., and Yong, J. (2010). Specify and enforce the policies of quantified risk adaptive access control. In *Proceedings of the 14th International Conference on Computer Supported Cooperative Work in Design*, pages 110–115.

Chen, L. and Crampton, J. (2011). Risk-aware role-based access control. In *Proceedings of the 7th International Workshop on Security and Trust Management*, pages 140–156.

Chen, L., Crampton, J., Kollingbaum, M. J., and Norman, T. J. (2012). Obligations in risk-aware access control. In *Proceedings of the Tenth Annual Conference on Privacy, Security and Trust*, pages 145–152.

Cheng, P.-C., Rohatgi, P., Keser, C., Karger, P. A., Wagner, G. M., and Reninger, A. S. (2007). Fuzzy multi-level security: An experiment on quantified risk-adaptive access control. In *Proceedings of the 2007 IEEE Symposium on Security and Privacy*, pages 222–230.

Cuppens, F., Cuppens-Boulahia, N., and Elrakaiby, Y. (2013). Formal specification and management of security policies with collective group obligations. *Journal of Computer Security*, 21(1):149–190.

Department of Defense (1985). *Trusted Computer System Evaluation Criteria*.

Elrakaiby, Y., Cuppens, F., and Cuppens-Boulahia, N. (2009). Formalization and management of group obligations. In *Proceedings of the IEEE International Symposium on Policies for Distributed Systems and Networks*, pages 158–165.

Elrakaiby, Y., Cuppens, F., and Cuppens-Boulahia, N. (2012). Formal enforcement and management of obligation policies. *Data & Knowledge Engineering,*, 71(1):127–147.

Garion, C. and Cholvy, L. (2007). Deriving individual obligations from collective obligations. In *Normative Multi-agent Systems*.

Graham, G. S. and Denning, P. J. (1972). Protection: principles and practice. In *Proceedings of*

the May 16-18, 1972, spring joint computer conference, AFIPS '72 (Spring), pages 417–429.

Grossi, D., Dignum, F., Royakkers, L., and j. Ch. Meyer, J. (2004). Collective obligations and agents: Who gets the blame. In *Proceedings of DEON04*, pages 129–145.

Harrison, M. A., Ruzzo, W. L., and Ullman, J. D. (1976). Protection in operating systems. *Commun. ACM*, 19(8):461–471.

Irwin, K., Yu, T., and Winsborough, W. H. (2008). Enforcing security properties in task-based systems. In *Proceedings of the 13th ACM Symposium on Access Control Models and Technologies*, pages 41–50.

ISO 10181 (1996). Information technology - Open Systems Interconnection - Security frameworks for open systems: Access control framework .

JASON Program Office (2004). Horizontal integration: Broader access models for realizing information dominance. Technical Report JSR-04-132, MITRE Corporation.

Kalam, A. A. E., Benferhat, S., Miège, A., Baida, R. E., Cuppens, F., Saurel, C., Balbiani, P., Deswarte, Y., and Trouessin, G. (2003). Organization based access contro. In *Proceedings of the 4th IEEE International Workshop on Policies for Distributed Systems and Networks*, pages 120–131.

Lampson, B. W. (1971). Protection. In *Princeton University*, pages 437–443.

Li, N., Chen, H., and Bertino, E. (2012). On practical specification and enforcement of obligations. In *Proceedings of the Second ACM Conference on Data and Application Security and Privacy*, pages 71–82.

Martinelli, F. and Morisset, C. (2012). Quantitative access control with partially-observable markov decision processes. In *Proceedings of the Second ACM Conference on Data and Application Security and Privacy*, pages 169–180.

Ni, Q., Bertino, E., and Lobo, J. (2010). Risk-based access control systems built on fuzzy inferences. In *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security*, pages 250–260.

OASIS (2005a). *Core and hierarchical Role Based Access Control (RBAC) profile of XACML v2.0*. OASIS. OASIS Standard (A. Anderson, editor).

OASIS (2005b). *eXtensible Access Control Markup Language (XACML) Version 2.0*. OASIS. OASIS Standard (T. Moses, editor).

OASIS (2005c). *SAML 2.0 profile of XACML v2.0*. OASIS. OASIS Standard (A. Anderson and H. Lockhart, editors).

OASIS (2007). *XACML v3.0 Obligation Families Version 1.0*. OASIS. OASIS Working Draft (E. Rissanen, editor).

OASIS (2010). *XACML v3.0 Core and hierarchical Role Based Access Control (RBAC) profile Version 1.0*. OASIS. Committee Specification (E. Rissanen, editor).

OASIS (2013). *eXtensible Access Control Markup Language (XACML) Version 3.0*. OASIS. OASIS Standard (T. Moses, editor).

Royakkers, L. and Dignum, F. (2000). No organization without obligations: how to formalize collective obligation? In *Proceedings of 11th International Conference on Databases and Expert Systems Applications*, pages 302–311.

Saltzer, J. H. and Schroeder, M. D. (1975). The protection of information in computer systems.

Proceeding of the IEEE, 63(9):1278–1308.

Sandhu, R. S. (1992). The typed access matrix model. In *Proceeding of IEEE Symposium on Research in Security and Privacy*, pages 122–136.

Sandhu, R. S. (1993). Lattice-based access control models. *Computer*, 26(11):9–19.

Sandhu, R. S., Coyne, E. J., Feinstein, H. L., and Youman, C. E. (1996). Role-based access control models. *IEEE Computer*, 29(2):38–47.

Thomas, R. K. (1997). Team-based access control (TMAC): A primitive for applying role-based access controls in collaborative environments. In *Proceedings of the Second ACM Workshop on Role-Based Access Control*, pages 13–19.

Thomas, R. K. and Sandhu, R. S. (1997). Task-based authorization controls (TBAC): A family of models for active and enterprise-oriented autorization management. In *Proceedings of the IFIP TC11 WG11.3 Eleventh International Conference on Database Security*, pages 166–181.