Pisa University

Faculty of Science
Computer Science Bachelor Degree

Internship Thesis

Support for Blacklist Policies in role based authorization system.

Candidate
Matteo Casenove

Supervisor
Prof. David Chadwick
University of Kent, UK

Referee
Prof. Roberto Barbuti
University of Pisa, IT

Academic Year 2010-2011

"Le grandi montagne hanno il valore degli uomini che le salgono, altrimenti non sarebbero altro che un cumulo di sassi." Cit. Walter Bonatti

"The great mountains have the valor of the men who climb them, otherwise they would be nothing but a heap of stones." Cit. Walter Bonatti

Abstract

This thesis describes a project carried out at the department of Computer Science at the University Of Kent in Canterbury. The professor David Chadwick, as chief of the Information Systems Security Research Group (*ISSRG*) at the UKC, proposed this project for an international exchange program called Erasmus Placement. The project was a part of another one, bigger than it, called TAS^3 (Trusted Architecture for Securely Shared Services) where the software *PERMIS* is developed. *PERMIS* (`http://sec.cs.kent.ac.uk/permis/`) is an infrastructure that provides all the necessary facilities for users to manage privileges and authorisation policies and for applications to make authorisation decisions.

The PERMIS PDP is a monotonic decision engine, in that Everyone is Denied Access except those that are allowed access by the rules. The addition of more rules grants more people access. There is no way of reducing accesses by adding more rules. The downside of a monotonic PDP is that it can be difficult to configure certain rules with exception clauses, for example, everyone at Kent has access to the library except Matteo. The work was focused on finding solutions to represent the exception rules: one of these requires to change the policy to Everyone is Granted Access Except, and the other requires to make a combined system of grant all PDP engine and the deny all PDP engine.

In this thesis we will discuss about *PERMIS* in general and how it works, we will described the policies used by *PERMIS* and then we will analyse the single parts of the project development of the new extension. Finally we will conclude with the results that the project has achieved and the possible future developments.

Acknowledgements

First of all, I would like to say thank you to the person who gave me the opportunity to live this experience and carry out this work, with availability and patience, even during the writing of my thesis: the Professor David Chadwick. A special thank goes also to Stijn, who helped me a lot during all my placement period. He was always fully available anytime and he has been a great colleague.

Ovviamente un grosso grazie va alla mia famiglia. In particolare ai miei genitori, senza i quali non sarei potuto essere qui, appoggiandomi sempre in ogni decisione e avendo pazienza anche nei momenti in cui ero meno promettente. Mammina, hai visto che l'hai "ricoldata"... Ai miei fratelli che ci sono sempre stati, partecipi in ogni mia decisione e difficoltá. Il fratello Paolo, un rifugio sempre accogliente e che in quanto a tecnologie non ci siamo mai fatti mancare niente, e Massi con prediche pallose e lunghe, con tanti "ci vuole che ti applichi", grazie, sono state sempre utili e aggiungerei "essenziali". A Massi e Cinzia e tutte le decisioni prese insime, se sono qui é anche grazie a voi (ps: ma non é ora che vi sposate? :)). A Stefy che é stato sempre un esempio di determinazione e voglia di fare e poi...riusciamo proprio a organizzare delle belle feste.

Un grande grazie a "via riminaldi" (Giuda, Lore', Bob, l'Orsetto, Frankie, Mary, il Beddo, Feffo, Pancio etc.), a quelli sempre presenti e a quelli di passaggio. Quanti momenti passati insieme e quante cazzate. Un hug particolare va al Gruppo Minix (Giuda e lore) che m'ha forgiato in questi anni d'universitá (dico solo "dal tramonto all'alba"). Abbiamo passato momenti fantastici.

Al professor Maestrini e alla professoressa Pelagatti, che hanno creduto in noi.

Agli amici di casa, agli amici di sempre. Grazie che m'avete fatto sentire sempre come se non fossi mai partito. A tutte le volte che visto che torno allora bisogna festeggiare, anche se comunque sia tornavo sempre. Quante ne abbiamo fatte e

quanti viaggi tremendi mi avete fatto fare per tornare a Pisa, ma comunque sempre con una storia da raccontare. Inutile fare la lista di tutti voi, ne siete troppi, ma un piccolo gruppo merita di essere nominato perché quello c'é sempre: Fra, Sandro, Alessandro e l'Harley.

In the end, I cannot avoid to mention the persons with which I shared most of the time that I spent on this work. The Canterbury friends. Starting from my colleagues, Kristy and Tom, and arriving to the London Road housemates, Alex, Laurence, Sian and Nicola. A special thank to Alex who has been by my side and helped me in any hard moment that I spent there. If I had success on this work is also thanks to you. All of you have given me an amazing period of my life.

Contents

Chapter 1

Introduction

This chapter provides an overview of the thesis and introduces the access control field. A presentation of the related technologies that will occur in this work, will be also provided.

1.1 Overview

Access Control is a very wide field and it is important in different environments. From enterprises to public services, whenever there are shared resources, it is used to ensuring a correct and authorised use of these resources. Access control is an important functionality for addressing security issues, especially now, when there is an exponential increasing trend of web applications.

PERMIS along with others, are *policy based authorisation infrastructures*. They permit to manage user privileges and they render access control decisions. Working at the software level, it permits any software to extend its access control features with the features provided by *PERMIS*. It can be remotely invoked by an application asking if a specific subject can perform a specific action on a specific resource. The system then will reply to the request with a proper response that contains *PERMIT* or *DENY*, according to the configured policy. The policies contain the rules and criteria that specify how user privileges are managed and access control decisions are made.

In accordance with the *RBAC* standard [RBAC], *PERMIS* uses whitelist policies which are specified as *permission* rules. When a request occurs in the

authorisation infrastructure, it will check if the request matches with an entry in the policy. If this happened, the authorisation infrastructure will reply with a *PERMIT* response. Otherwise, it will reply with either a *DENY* or *NOT APPLICABLE* response.

This is a monotonic decision engine that uses a Deny All Policy in which are specified subjects who have an access guaranteed. In contrast, it can be difficult to configure the exception clauses in the rules , e.g., *"All students are granted access except MSc students"*. To represent the previous rule, it is necessary to either modify the current *PERMIS* policy to accept the exception clauses or create a blacklist with the exception clauses controlled by a new reversed PDP, and combine this second PDP with the original one. Both of these solutions have been implemented in the course of the project and will be presented in this thesis.

However, before describing all the project, an introduction of all the aspect behind *PERMIS* is necessary to fully understanding it.

1.2 The Access Control model

In computer security, access control includes three subfields: the authentication phase recognise the entity that wants to use the system and then, with the authorisation phase, the system checks if the entity has the rights to perform one specific action and according to that it gives the authorisation. In the end, with the audit phase the system logs the operations performed to keep track of them.

There are various Access Control Models but can be divided in two categories: discretionary or non-discretionary[ACW]. In the latter, there is a security policy administrator who is the only one that has right to give and configure access permissions for all the resources. On the other hand, the discretionary model allows the user to make policy decisions and/or assign security attributes for his own resources. The three most accredited models are: *Discretionary Access Control (DAC)*, *Mandatory Access Control (MAC)* and *Role Based Access Control (RBAC)*. In *DAC* only the owner decides who is

allowed to gain access to the object and which privileges they have. In *MAC* there is a rule-list specified by the administrator and the access is permitted, if and only if there is a rule that allows a given user to gain access to a resource. Very similar to the *MAC* model is the *RBAC* model. It is not necessarily a mandatory model, but could be also a discretionary one according to the needs of the system. It is based on the concept of role that can be assigned to user and then is the role that has the permission of performing the action. It has a level of indirection, the indirection being the role between the user and the permission. *ABAC* is a generalization of the *RBAC* model in which a role is not restricted to an organizational role, but can be any attributes of the subject.

PERMIS uses the *ABAC* (then also *RBAC*) model, in which roles are used to model organization roles, user groups, or any attributes of the user.

1.3 Role-Based Access Control

Very important for us is to analyse in detail the *RBAC* model, since *PERMIS* is built on this model.

The *RBAC* (Role-based Access Control) [RBAC] provides a popular model for information security, designed to address many needs of the commercial and government sector. Several studies indicate that permission assigned to roles tend to change relatively slowly compared to changes in user membership of roles. For this reason, *RBAC* introduces the concept of role in the Access Control, defining role-permission instead of user-permission relationships. In this way, the user is just assigned to the predefined roles. It results an easier procedure compare to create associations for each user to permissions.

It is policy neutral, it directly supports three well-known security principles: least privilege, separation of duty, and data abstraction. Least privilege is supported by assigning to the role only the minimum permissions required for the tasks conducted by the members of the role. Separation of duties is achieved by ensuring that mutually exclusive roles must be invoked to complete a sensitive task. Data abstraction is supported by means of abstract

permissions such as open or close for a door rather than read or write for a file.

There is no a well defined standard for the RBAC model provided by NIST [NISTRBAC] but it leaves many aspects undefined and this leads on to have a large scale of different interpretations and implementations of the model. Also the *NIST* model of *RBAC* has inside different levels of it with different features for each level: Flat RBAC, Hierarchical RBAC and Constrained RBAC.

1.3.1 Flat RBAC

Flat RBAC contains the essential concept of the whole model. The figure 1.1 shows three sets of entities called *user* (U), *role* (R) and *permissions* (P). The basic concept of it is that users are assigned to roles, permissions are assigned to roles and users acquire permissions by being a member of roles. The standard requires that user-role and role-permission assignment are many-to-many. This is an essential aspect of *RBAC*.

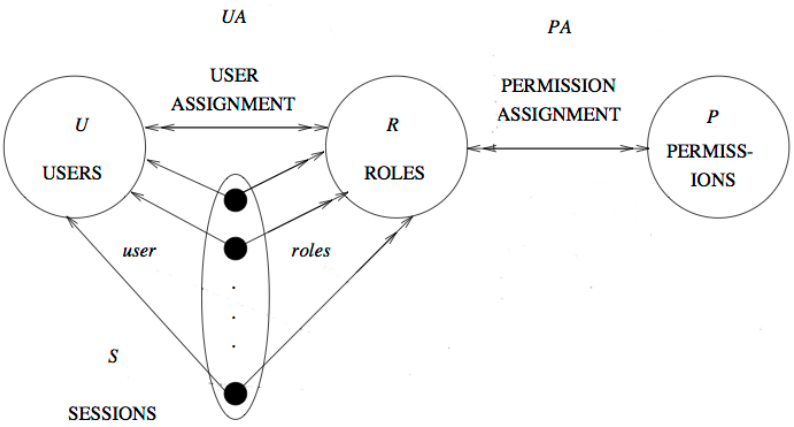

Figure 1.1: Flat RBAC

In this model the User is a human being but an abstraction of this can be accepted such as an autonomous system. A Role is a job function or job title within the organization with some associated semantics regarding the authority and responsibility conferred on a member of the role. A Permission is an

approval of a particular mode of access to one or more objects in the system[1]. Permissions are in general positive and confer the ability to the holder to perform some action(s) in the system[2]. The nature of the permissions greatly depends on the implementation details of a system and its type. RBAC has no restrictions of the meaning on the permissions, treating them as uninterpreted symbols to some extent. In the flat RBAC there is also the concept of sessions. Each *session* is a mapping of one user to possible many roles, i.e., a user establishes a session during which he "activates" some subsets of roles that he or she is a member of. The permissions available to the user are the union of permissions from all roles activated in that session.

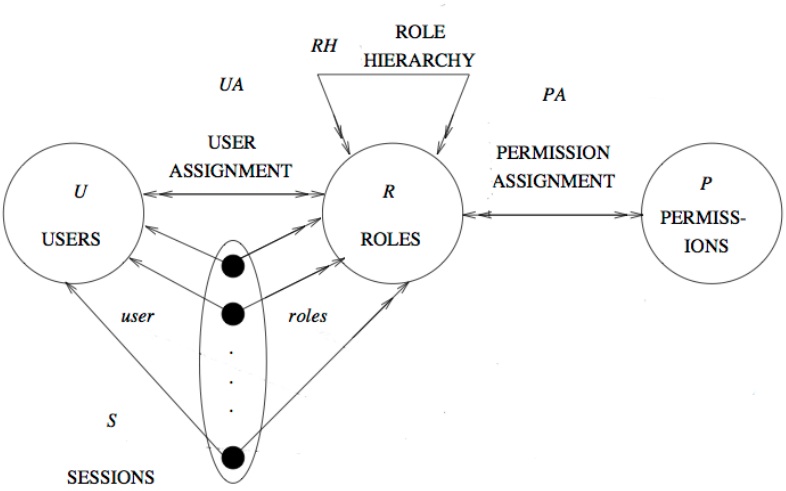

Figure 1.2: Hierarchical RBAC

A user may have multiple sessions opened at the same time and each session may have a different combination of active roles. In particular, powerful roles can be kept dormant until they are needed. In this way the least privilege principle is supported by RBAC.

Many issues are left open by the flat RBAC standard and they are differently managed from implementation to implementation, such as the behaviour

[1]The term authorization, access right and privilege are used in literature with the same meaning of permission

[2]Flat RBAC does not exclude the negative permission which deny access. In fact, using this freedom, our project extend the normal RBAC policy of PERMIS with negative rules

of revocation and the use of condition.

1.3.2 Hierarchical RBAC

The second level of RBAC includes roles hierarchies, see Figure 1.2. Every time that the roles are discussed, the hierarchies are discussed too, being the natural interpretation of the organization's lines of authority and responsibility always present in the real situations. An example is on the Figure 1.3. Mathematically, these hierarchies are partial orders.

By convention more powerful (or senior) roles are shown toward the top of role-hierarchy diagrams, and less powerful (or junior) roles toward the bottom. Senior roles inherit all permissions of their junior roles respectively, and this inheritance of permission is transitive too (this is true as long as the hierarchy is a partial order). The inherited permissions are added to the own role permissions. In the example of Figure 1.3, Staff and Researcher inherit the Student permissions but each one of these have different permissions directly assigned to it, so in the end they have a different set of permissions.

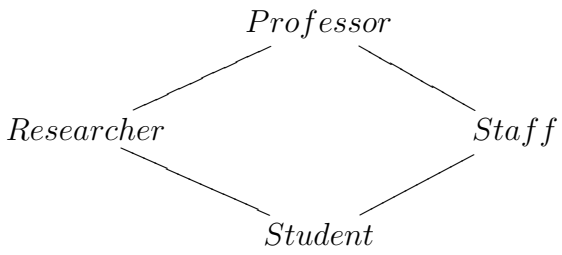

Figure 1.3: Example of Role Hierarchy

1.3.3 Constrained RBAC

Constrained RBAC adds constraints to the hierarchical RBAC model. They are predicates which, applied to relationship, return a value of "acceptable" or "not acceptable". The constraints may be applied to the user-role assignment, to the role-permission assignment, or to the activation of roles within user sessions. They can be applied in a lot of different ways, so we will just discuss some some interesting constrains.

The most frequently mentioned constraint is *mutually exclusive* roles. The user can be assigned to at most one role in a mutually exclusive set. This supports separation of duties. The mutually exclusive role can be also applied to the role-permission relationship. It specifies that the same permission cannot be assigned to two roles. Of course these two constraints can be applied at the same time requiring that the same permission cannot be assigned to more than one user in a mutually exclusive set.

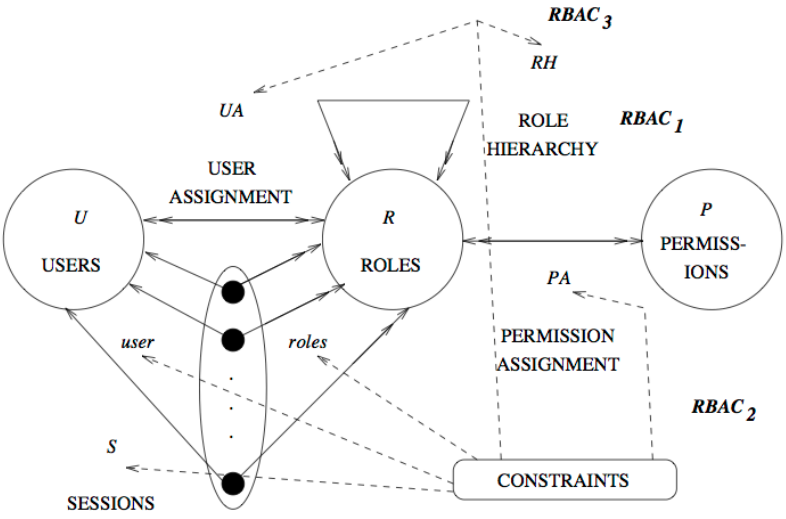

Figure 1.4: RBAC models

Another example of user assignment constraint is that a role can have a maximum number of members. It is called *cardinality constraint*.

Constraints can also be applied to sessions, and the user and roles functions associated with the session. For example it may be acceptable for a user to be a member of two roles but the user cannot be active in both roles at the same time.

This supports dynamic separation of duties.

Somehow also the hierarchy can be considered as a constraint. The constraint is that a permission assigned to a junior role must also be assigned to all senior roles.

All the levels of RBAC that we have presented can be merged together to combine the different features as in the Figure 1.4.

We have introduced a family of RBAC models that can be applied in almost every environment. We saw that RBAC is a very powerful method in the Access Control field but is not a panacea for all access control issues. More sophisticated forms of access control are required to deal with situations where sequences of operations need to be controlled. RBAC does not attempt to directly control the permissions for such a sequence of events. Other forms of access control can be layered on top of RBAC for this purpose.

1.4 Related Technologies

PERMIS includes different type of support to increase its portability and adaptability in different environment. We will make a list of the most important technologies used by *PERMIS* and we will describe how they are used.

- ***LDAP*** stands for *Lightweight Directory Access Protocol* and it is an application protocol for accessing and maintaining distributed directory information services over an *Internet Protocol (IP)* network. It defines a directory service information and query models. The information model is centered around entries, which are composed of attributes. The entries are organized into a tree structure, usually corresponding to a geographical and organizational distribution. The query models allow searching of portions of the tree based on filter criteria involving attributes, and returning requested attributes from each matching entry [LDAP].

  It is used by *PERMIS* as a network accessible repository for storing policies and credentials.

- ***X.509*** is a standard for a *Public Key Infrastructure* (*PKI*) and *Privilege Management Infrastructure* (*PMI*). *X.509* specifies, amongst other things, standard formats for public key certificates, certificate revocation lists, attribute certificates, and a certification path validation algorithm [X509].

PERMIS uses it to provide trust and tamper-proof resistance to policies and credentials. But along with *X.509*, *PERMIS* can use other formats including plain *XML* policies and *SAML* attribute assertions.

- ***SAML*** is an *XML-based* open standard for exchanging authentication and authorization data between security domains. It uses security tokens containing assertions to pass information about a principal (usually an end-user) between an identity provider and a web service. *SAML 2.0* enables web-based authentication and authorization scenarios [SAMLW].

  PERMIS supports *SAML* assertions in the requests.

- ***XACML*** is an *XML-based* language for access control that has been standardized in *OASIS*. *XACML* describes both an access control policy language and a request/response language. The policy language is used to express access control policies (who can do what when). It provide a mechanism that offers much finer granular access control than simply denying or granting access – that is, a mechanism that can enforce some before and after actions (called obligations) along with "permit" or "deny" permission [SUNXACML] [IBMXACML] [OASISXACML].

  PERMIS implements a *XACML* interface to receive *XACML* request and send *XACML* response, but *PERMIS* does not support the *XACML* policy language, as it has its own *RBAC* based language.

From now on, we will describe the internal *PERMIS* implementation and how it is used in a real environment. We will explain a PERMIS policy and how to write a correct one. The last chapter will illustrate the design and the consequent implementation of the new extension for the blacklist support.

Chapter 2

PERMIS

In this part will be presented the whole *PERMIS* infrastructure, how it works and how it interacts with other services in a real environment.

2.1 Introduction

In the Access Control field, as we have already said, there are three main phases: *Authentication*, *Authorisation* and *Audit*. The first determines who the user is, the second what he is allowed to do and the last one keeps track of what he did. *PERMIS* is an Authorisation Infrastructure that manages privileges and makes access control decisions. Its functionality goes over the normal authorisation system. It implements the novel concept of credential validation which verifies user's credentials permitting then to use a distributed management of credentials. *PERMIS* also supports delegation of authority, thus credentials can be delegated between users, further decentralizing credential management. All these characteristics permit to *PERMIS* to easily adapt to every heterogeneous system and especially in distributed systems.

2.2 PERMIS Access Control Infrastructure

PERMIS has the policies at the base of all the authorization system, to provide flexibility, scalability and application independency. The authorisation model paradigm adopted is the well-known "Subject - Action - Target" paradigm enhanced with the *ISO* Attribute Based Access Control ($ABAC$) model [ABAC].

Unfortunately, in a distributed environment, attributes travel between different independent systems and a malicious user could claim attributes which he is not the rightfully owner. Consequently, subject's attributes are presented as digitally signed credential issued to the subject by one or more trusted attribute authorities (AAs)[1]. Once that the credentials are digitally signed, the system has to check if they are original and issued by trusted AAs. *PERMIS* introduces a Validation Service called Credential Validation System (CVS) to validate these credentials. The CVS uses policies to specify the trusted AAs and the rightful attributes of the user; each resource owner specifies the credential validation policies for gaining access to his resources.

PERMIS uses the Hierarchical $RBAC$ (or $ABAC$) model, in which roles are used to model organization roles or any attributes of the user. Roles or attributes may be organized in a partial hierarchy. A superior role inherits all the privileges allocated to its subordinate roles. Role hierarchies do not need apply only to organizational roles, but can apply to any attributes, such as level of authentication (LoA), where there is a natural precedence in the attribute values, in which higher value implies the privileges of the lower values.

An Authorisation Infrastructure (AI) is placed between the subject and the target. Conceptually, it receives the request from the subject and it checks if the request is correct and if the subject is authorised to perform the requested action on the requested target. If AI authorises the request, it is delivered to the target. Otherwise it is rejected.

Figure 2.1 shows our high level conceptual model for an authorisation infrastructure. The numbers sign the steps that the system performs to process a request. Step 0 is the initialization step for the infrastructure, when the policies are created and stored in the various components. When a subject issues an application request (step 1), the application policy enforcement point (PEP) includes the user credentials in the request, collecting them from the Credential Issuing Service (CIS) or Attribute Repository (AR) (steps 3-4).

[1] AAs have the aim to issue Attribute Certificates (ACs). They also provide the service to revoke ACs, keeping the revoked AC in a Attribute Certificate Revocation List ($ACRL$). Since issued, ACs and $ACRLs$ can be stored in a directory system; user can obtain them by using $LDAP$.

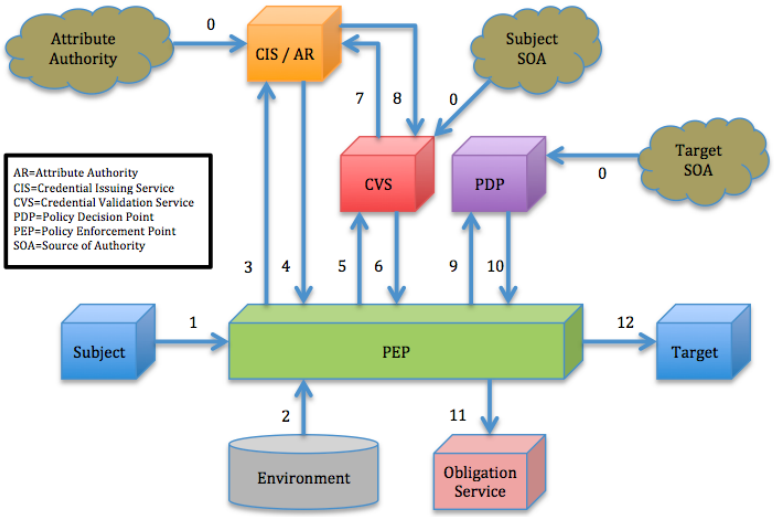

Figure 2.1: Authentication Infrastructure Model

After that, the credentials are validated (step 5) by the Credential Validation Service accordingly with the policy. The valid credentials are returned to the *PEP* (step 6), combined with the environmental information, such as current time and date (step 2), and then passed to the PDP along with the request for an access control decision (steps 9-10). If the *PDP*, according to the policy, grants the request, then this is allowed to reach the target (step 12), otherwise it is rejected. In either case, along with the response the *PDP* may return a set of obligations, which are actions that the *PEP* must enforce in order to complete the authorisation (step 11). The Obligations Service is the functional component responsible for enacting these obligations.

These steps are the effective interactions between the software and the Authorisation Infrastructure and between the services inside the infrastructure itself. In the conceptual model the Subject box represents the interface with the user and the Target is the interface of the requested resource. The *PEP* is a procedure implemented in the software. It is the enforcing point and it interacts directly with the external services such as the *PDP* and the *CVS*. Both are the core of the *AI*, implementing the main functionalities. The *PDP* will be described in detail in one of the next chapters.

In this distributed scenario it is necessary to have, behind the authori-

sation infrastructure, a Trust Infrastructure to permit that everything works correctly. Credentials are the format used to securely transfer a subject's attributes/roles from the Attribute Authority to the recipient. They are also known as attribute assertions. They can also be allocated in form of *X.509*Attribute Certificates (ACs[2]) digitally signed and usually stored in either a local file or *LDAP* directory. These bind the issued attributes with the subject's and issuer's identities in a tamper-proof manner. *PERMIS* only trusts valid credentials issued by trusted AAs or their delegate in accordance with the current policies in the authorisation infrastructure (Issuing, Validation and Delegation policies).

But how is a trusted AA recognised? It is a job of the *PERMIS*'s Credential Validation Service. The *CVS* checks that each credential issuer is mentioned in its policy. This policy contains rules that govern which attributes different AAs are trusted to issue, along with Delegation Policy for each AA.

Trivially the validity of the credential is proved by its digital signature from the AAs. However, this requires a trusted *PKI* to be implemented to support *PERMIS*, as *PERMIS* does not provide certificate verification itself.

2.3 The PERMIS delegation system

Often in real situations it happens that a person asks to someone else to do his job for a short time. For example, a boss could ask his employee to perform his task, so he "delegates" to him the task. In this way the employee has the authorisations of the boss level (role) for the time that the task needs, and after that, he will be no longer be authorised at that level. The boss gives the authority to the employee to perform the task on his behalf.

This situation is supported in the *PERMIS* authorisation infrastructure by a Delegation Issuing Service (DIS). It is a web service that dynamically issues ACs on demand when requested to by a delegator. The delegator delegates his attribute(s), i.e. his role(s), to another user to permit the latter to have the

[2]A set of attributes and a public key certificate identifier that are made unforgeable by use of the digital signature created with a private key.

right permissions to perform the task on the delegator's behalf. Obviously a delegation can be revoked.

The delegation is completely independent of any administrative involvement, in fact it may be called directly by any application's PEP to issue short lived ACs. The DIS is controlled by its own policy that specify who is allowed to delegate what to whom.

A delegate can be authorised to yield his role in turn to others, but each AA may constrain delegations by validity times and delegation chain lengths. In this way for example a delegation can be reused maximum 2 times and it will expire in a week. *PERMIS* ensures that all delegated credentials conform to the following delegation paradigm:

i) an issuer cannot delegate more privileges then he possesses, and

ii) an issuer cannot delegate a privilege to himself or to a superior in the delegation chain, since the recipient already holds this privilege.

Following these two rules the DIS avoids the propagation of privileges from issuers to subjects and the inappropriate delegation that could remove the controls of the delegations.

2.4 The PDP

The *PERMIS* Authorisation Decision Engine is responsible for credential validation and access control decision making. Credential validation is the process that enforces the trust model as described in Section 2.2, and ensure that only valid roles/attributes are attributed to users. Access Control decision making instead, is the process that ensures only users with the required attributes gain access to the protected resources. Both of them are important, but only the Access Control is essential for an authorisation infrastructure.

The component *PDP* is responsible for making access control decisions based on the valid attributes of the user and the Target Source of Authority[3] access control policy, which is a subset of the *PERMIS* policy.

[3]A Source of Authority or *SOA* is the root of trust of a Privilege Management Infrastructure. This is an entity that a resource implicity trusts to allocate privileges and access

The *PDP* has inside a Policy Parser for each type of policy. It is one of the most important part of the *PDP*. It has the several aims: it checks the validity of the policy, it reads the policy and in the end it loads all the policy information. The parser can read plain *XML* or digitally signed and protected policies. The former type of policies are stored as text file in the local filestore whilst the latter are stored as *X.509* policy *ACs* in either the local filestore or the Target *SOA's* entry in an *LDAP* directory. The difference between the two type of policies is the type of protection: the normal *XML* policy being stored as normal text file is protected by the *OS*, on the other hand the *X.509 ACs* are tamper resistant and integrity protected by cryptography.

Once the Parser has processed the policy[4] the *PDP* is ready to accept the request. Its core is in the decision engine, the delicate part that makes decision according to the parsed policy. The decision engine takes the request composed of the Subject of the request, the Action to perform, the Target on which the Subject wants to perform the Action and the eventual environmental attributes. Accordingly with the policy it checks each element of the request and if all matches then it returns "PERMIT" as reply, "DENY" or "NOT APPLICABLE" in the other case. The *PEP* is in charged of passing the request to the *PDP* along with user's valid attributes and any required environmental attributes. But the environmental attributes do not depend on the target of the request or the action to perform and so, they are always the same. This means that, before to be included in the request the environment attributes have to be retrieved by the *PEP* somehow.

When the PDP checks the request with the policy different situations can occur: i) the policy has a rule for the request, ii) the policy has NOT a rule for the request, iii) the policy has a rule but something is missing from the request and iv) the policy has a rule for some information of the request but not for all. In i) the PDP simply finds the rule that matches with the request and it returns PERMIT as reply. In ii) on the other hand there are no rules that grant the request, so the *PDP* returns DENY. In iii) there is a more

rights to it. The *SOA* is ultimately responsible for issuing *ACs* to trusted holders, and these can be either end users or subordinate *AAs*.

[4]Every time the policy changes, the parser re-pars it to load the new changes.

sophisticated scenario where the decision engine finds a rule to match but the rule requires more attributes then the request has, such as more environmental attributes. In this case the *PDP* returns DENY or INDETERMINATE as the reply depending upon what is missing. The case iv) could happen when the request is out of the scope of the policy, for example when the action or the target in the request are not in the domains. In this particular case the *PDP* replies with NOT APPLICABLE.

These are the responses returned to the *PEP*, but actually the *PERMIS APIs* return exceptions in case of INDETERMINATE o NOT APPLICABLE situations. These exceptions are then wrapped in *XACML* responses and than returned.

The request can provide also extra information such as additional roles or environmental variables, and in this case the extra information is ignored.

These replies return to the *PEP* that, accordingly with their values, rejects the request, delivers the request or communicates the error to the user.

This description of the infrastructure and how the various components interact is enough to have an idea of the authorisation infrastructure and how it works. The trust model and the decision engine are very important in the infrastructure. The first one permits to connect all the components located in a distributed system keeping security and trustworthy in every steps whereas the second one makes the decisions necessary to control the accesses.

Chapter 3

The policies

As we said before, policies are at the base of the *PERMIS* authorisation infrastructure. They give the guidelines for the actions of the components.

In this chapter we will present the *PERMIS* policies and we will describe how to write a correct one. We will also present two softwares implementations around *PERMIS* to manage policies: the Policy Editor and the Policy Tester. They are important resources for an administrator, providing tools to write policies easily and to easily test them.

3.1 Introduction to Policies

In *PERMIS*, the policies implement the *RBAC* authorisation model. The characteristics of a policy are: what it defines, how it is written and how it is represented.

Implementing *RBAC*, the policy defines the authorisation rules. *PERMIS* specifies an *RBAC* policy specifically designed for use with an *X.509* attribute certificate based PMI[1]. X.509 supports RBAC by defining *role specification attribute certificates* that hold the permission granted to each role, and *role assignment attribute certificates* that assign various roles to the user. In *role specification ACs*, the holder is the role , and the privilege attributes are permissions granted to the role. In *role assignment ACs*, the holder is the user[2], and the privilege attributes are the roles assigned to the him. The

[1]Privilege Management Infrastructure

[2]The name of the user

user is identified by either his *LDAP Distinguished Name (DN)* or his public key certificate (issuer and serial number). The *PERMIS X.509 PMI RBAC* Policy is the union of a number of sub-policies as shown in Figure 3.1. The domain of top level policy is the union of the sub-policies domains. Each of them has a *unique object identifier (OID)* to globally identify it. Passing the *OID*, *PERMIS* can retrieve the correct policy to use for making access control decisions.

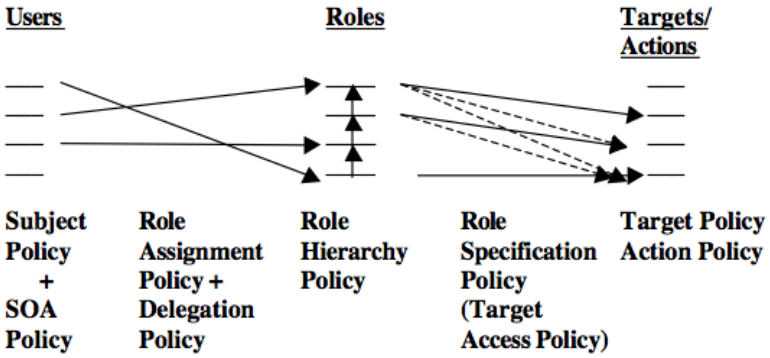

Figure 3.1: The X.509 PMI RBAC Policy and its Sub-Policies

The policy is written in *XML* to provide an easy way to define policy by administrators and to make the policy easy to read by users. *PERMIS* provides its own *schema*[3] to define the way to write the policy [SCHEMA]. The schema is a meta-language that holds the rules for creating the *XML* policies. Each rule of the schema specifies how to write a rule in the policy and which statements have to be used to write it. The main components of our schema are the following:

- **SubjectPolicy** - this specifies the subject domains. Only users from a subject domain can be authorised to access resources covered by the policy.
- **RoleHierarchyPolicy** - this specifies the hierarchy between different

[3]An XML schema is a rigorous specification of an XML-based language in terms of constraints on elements and attributes. An XML document is said to be valid against a schema if the elements and attributes in this XML document satisfy the constraints specified in the schema.

roles.

- **SOAPolicy** - this specifies which SOAs are trusted to issue roles.
- **RoleAssignmentPolicy** - this specifies which roles may be allocated to which subjects by which *SOAs*.
- **TargetPolicy** - this specifies the target domains. Only targets in the domain can be accessed by a user.
- **ActionPolicy** - this specifies the actions (or methods) supported by the targets.
- **TargetAccessPolicy** - this specifies which roles are authorised to perform which actions on which targets, and under which conditions. Conditions are specified in Boolean logic and may contain constraints. All the actions that are not present in the Target Access Policy are denied.

The following figures show some main parts of the schema.

```
[...]
<xs:element name="SubjectPolicy">
    <xs:complexType>
      <xs:sequence minOccurs="1" maxOccurs="unbounded">
        <xs:element ref="SubjectDomainSpec" />
      </xs:sequence>
    </xs:complexType>
    <xs:key name="SubjectDomainSpecKey">
      <xs:selector xpath="./SubjectDomainSpec"/>
      <xs:field xpath="@ID"/>
    </xs:key>
  </xs:element>

  <xs:element name="SubjectDomainSpec">
    <xs:complexType>
      <xs:complexContent>
        <xs:extension base="DomainSpecType">
          <xs:attribute name="ID" type="xs:string" use="required"
              />
        </xs:extension>
      </xs:complexContent>
    </xs:complexType>
  </xs:element>
```

```
  <xs:complexType name="SubtreeDefType">
    <xs:annotation>
      <xs:documentation>
        SubtreeDef is the abstract base type of Include and
            Exclude elements
                as specified by Sassa Otenko
      </xs:documentation>
    </xs:annotation>
    <xs:attribute name="LDAPDN" type="xs:string">
      <xs:annotation>
        <xs:documentation>
          For PERMIS LDAP DNs are equivalent to simply DNs. Both
              formats are equally acceptable to PERMIS. Exclude is
              used to exclude subtrees from within an included
              subtree LDAPDN is an LDAP DN from RFC 2253 or a
              simple DN. CN=guest,OU=GlobusTest,O=Grid means the
              same as /CN=guest/OU=GlobusTest/O=Grid.
Max and Min have the same semantics as for the Include subtree
    specification. Note that either DN or URL must be present (
    unlike Include, where both may be missing). The semantics of
    the DN and URL are the same as for Include.
Semantic: LDAPDNs cannot be fully described by the Schema so a
    software check is required for this
        </xs:documentation>
      </xs:annotation>
    </xs:attribute>
    <xs:attribute name="URL" type="xs:anyURI">
      <xs:annotation>
        <xs:documentation>
          Semantic: A check is necessary to make sure this URI
              conforms to a supported URI scheme
        </xs:documentation>
      </xs:annotation>
    </xs:attribute>
    <xs:attribute name="Min" type="xs:nonNegativeInteger"/>
    <xs:attribute name="Max" type="xs:nonNegativeInteger"/>
  </xs:complexType>

<xs:element name="Include">
    <xs:complexType>
      <xs:annotation>
        <xs:documentation>
          Subject Domain must contain at least one LDAP sub-tree.
              We do not support single entries at the moment. (So
              if a new sub-node is created, and it is not in the
              Exclude statement, it will be allowed.)
        </xs:documentation>
      </xs:annotation>
      <xs:complexContent>
        <xs:extension base="SubtreeDefType">
          <xs:sequence maxOccurs="unbounded">
```

```
            <xs:element minOccurs="0" ref="Exclude"/>
          </xs:sequence>
        </xs:extension>
      </xs:complexContent>
    </xs:complexType>
  </xs:element>

  <xs:element name="Exclude" type="SubtreeDefType"/>
[...]
```

Figure 3.2: Subject element of the schema for the PERMIS policies

As we can see in Figure 3.2, with the *SubjectPolicy* we define it as a *complexType* composed by a *sequence* of *element* and that the *element* is a *SubjectDomainSpec*. The latter is specified below the *SubjectPolicy* in almost the same way and so on.

In this way the schema defines that a *SubjectPolicy* statement has as children one or more *SubjectDomainSpec* statements, each one with an attribute *ID* with type string (obligatory). In turn, the *SubjectDomainSpec* has one or more *Include* or *Exclude* statements, each one with an attribute LDAPDN with type string (obligatory).

```
[...]
<xs:element name="TargetAccessPolicy">
    <xs:complexType>
      <xs:sequence maxOccurs="unbounded">
        <xs:element ref="TargetAccess" />
      </xs:sequence>
    </xs:complexType>
  </xs:element>

  <xs:element name="TargetAccess">
    <xs:annotation>
      <xs:documentation>
        The target access policy comprises one or more target
            accesses. Each TargetAccess allows an initiator with
            the specified set of roles to carry out the specified
            actions on the list of targets, but only if the
            conditions specified by the optional IF clause are true
            , and only if the initiator is not attempting to
            perform a conflicting action as specified in the MSoD
            policy. The initiator must possess all of the roles in
            the RoleList in order to get access. If access is
            granted, any specified obligations will be returned to
            the PEP
```

```
      </xs:documentation>
    </xs:annotation>
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="RoleList" />
        <xs:element ref="TargetList" />
        <xs:element minOccurs="0" ref="IF" />
        <xs:element minOccurs="0" ref="Obligations" />
      </xs:sequence>
      <xs:attribute name="ID" type="xs:string">
        <xs:annotation>
          <xs:documentation>TargetAccess ID is a label for humans
              and is ignored.
                 </xs:documentation>
        </xs:annotation>
      </xs:attribute>
    </xs:complexType>
  </xs:element>

  <xs:element name="TargetList">
    <xs:complexType>
      <xs:sequence>
        <xs:element maxOccurs="unbounded" ref="Target" />
      </xs:sequence>
    </xs:complexType>
  </xs:element>

  <xs:element name="Target">
    <xs:annotation>
      <xs:documentation>
        Target specifies a target instance or domain and the
            actions that can be carried out on it
        Changes: Actions attribute that contained the comma
            separated list of actions allowed is now a sequence of
            one or more AllowedAction elements
      </xs:documentation>
    </xs:annotation>
    <xs:complexType>
      <xs:sequence>
        <xs:choice>
          <xs:element ref="TargetName" />
          <xs:element ref="TargetDomain" />
        </xs:choice>
        <xs:element ref="AllowedAction" minOccurs="0" maxOccurs="
            unbounded"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
[...]
```

Figure 3.3: Target Access elements of the scheme for the PERMIS policies

Another significative statement is the *TargetAccessPolicy* (Figure 3.3) and it is defined as a sequence of *TargetAccess*. In the schema the *minOccur* attribute for the sequence of *TargetAccess* is not present so there may not be any of them defined. If this happens, no access rule is defined, and so nobody can access any resources. Apart from that, *TargetAccess* is defined as a sequence of four elements: *RoleList*, *TargetList*, *IF* and *Obligations*. The first is the list of roles that is authorised to perform the action. The *TargetList* is in turn a sequence of *Target* composed by a *TargetName* or a *TargetDomain* and an *AllowedAction*. With the *IF* statement the schema introduces the constraints whilst the *Obligations* define the actions that the system has to perform as a consequence of the action in the request.

Along with these, other statements are defined in the schema to define the way to write the *PERMIS* policies. The previous two figures are just some pieces of the schema. The complete version of it can be found on the *PERMIS* site.

In the end, the *PERMIS* policy can be represented either as a normal plain *XML* file stored in the filesystem or as *X.509 AC* stored in the filesystem or in a *LDAP* directory.

3.2 Policies structure

The structure of the policy follows the components dictated in the schema and described above. In fact, the schema defines the correct way to write the policy and imposes the exact steps to follow. Only a policy that respects completely the schema can be accepted. This procedure is called validation and it is in charged of checking if the policy is written following the rules provided by the schema.

A policy can be divided in different parts where each one describes a particular aspect of the situation we are representing. Each part of these will be presented individually.

3.2.1 Subject Policy

The Subject Policy is the first part of the policy. It is always at the beginning and it specifies the domains of the users who can be granted roles. The user is identified by his *LDAP DN* and each domain is specified as an *LDAP* subtree, using Include and Exclude statements. The Include statement allows to specify the *LDAP DN* of the root node of a subject domain. On the other hand, the Exclude statement excludes the *LDAP* subtree from the domain. Both Include and Exclude statements have two optional attributes (Min and Max) to provide layering in the inclusion or exclusion. With Min and Max, it is possible to specify the depth of the layers of inclusion or exclusion respectively. The default value of Min is zero, meaning the root of the subtree, and the default for Max is infinity, meaning the leaves of the subtree. If the *LDAP DN* is "null" in an Inclusion statement, it means that the domain is all the users in the world.

Let's suppose we have a subject domain that includes all users at the University of Pisa and only researchers at the University of Kent, but that excludes the department of engineering of the University of Pisa. Let's suppose also to have another subject domain composed of the managers of Sun Microsystems. The resulting Subject Policy can be written as below.

```
<SubjectPolicy>
   <SubjectDomainSpec ID="SunMicrosystemsManagers">
      <Include LDAPDN="ou=managing,dc=SunMicrosystems,c=us"/>
   </SubjectDomainSpec>
   <SubjectDomainSpec ID="ResearchDomain">
      <Include LDAPDN="dc=unipi,dc=it"/>
      <Include LDAPDN="ou=research,dc=unikent,dc=uk"/>
      <Exclude LDAPDN="ou=engineering,dc=unipi,dc=it"/>
   </SubjectDomainSpec>
</SubjectPolicy>
```

Figure 3.4: Subject Policy Example

If a subject requires the access but is not specified in the Include statements as *LDAP DN* or in its subtrees, the system will deny the access and a *Subject Out Of Domain* exception will be thrown.

3.2.2 SOA Policy

The Source of Authority (*SOA*) Policy has the aim to specify who is trusted to issue roles to the subjects specified in the subject policy. It is a list of *LDAP DNs* of the *SOAs*. These *DNs* will match the root issuer name in published *ACs*.

The first *DN* in the list has to be the policy creator's *DN*, and it must be always present. Subsequent, names of the possible remote *SOAs* have to be present to ensure that the *ACs* signed by them are accepted. In other words, every *AC* that is to be trusted by the policy, must have been signed by one of the *SOAs* in the list. If this does not happen, the *AC* is not accepted.

Some cases require a *SOA* list composed by only the policy creator, and consequently all *ACs* will have to be signed by him. In other cases external *SOAs* may be necessary, providing then a support of externally allocated privileges.

```
<SOAPolicy>
    <SOASpec ID="SOA" LDAPDN="cn=MatteoCasenove, ou=student, o=
        unipi, c=it" />
</SOAPolicy>
```

Figure 3.5: SOA Policy Example

The above *SOA* Policy code defines a single *SOA* represented by the student Matteo Casenove that has to be the policy creator, too.

3.2.3 Role Hierarchy Policy

The Role Hierarchy Policy defines the role hierarchies that are supported by the *RBAC* policy. Each role hierarchy (the RoleSpec statement in the schema) is a directed graph whereby it supports multiple superior roles inheriting the privileges of a common subordinate role and also that a superior role inherits all the privileges of a set of subordinate roles.

Each role is represented as an Attribute Type, Attribute Value pair where the first is the type of the role and the second is the value (name) of the role. Generally, two elements must be of the same type to be able to relate to each other. Accordingly, the roles, member of the same hierarchy, must be of the same type, usually it is the *LDAP* attribute type name.

In the example below, we consider a basic hierarchy of a company composed by the Managers at the top, as its subordinate the Finance Department and the Marketing Department and at the bottom, as subordinate of them, there is the Employee.

```
<RoleHierarchyPolicy>
        <RoleSpec OID="1.2.826.0.1.3344810.1.1.20" Type="
            companyRole">
            <SupRole Value="Employee"/>
            <SupRole Value="Manager">
                <SubRole Value="FinanceDepartment"/>
                <SubRole Value="MarketingDepartment"/>
            </SupRole>
            <SupRole Value="FinanceDepartment">
                <SubRole Value="Employee"/>
            </SupRole>
            <SupRole Value="MarketingDepartment">
                <SubRole Value="Employee"/>
            </SupRole>
        </RoleSpec>
    </RoleHierarchyPolicy>
```

Figure 3.6: Hierarchy Policy Example

3.2.4 Role Assignment Policy

Once we have defined the subject domain and the roles, it is necessary to specify how to assign roles to the subjects. This is the duty of the Role Assignment Policy, specifying which roles can be assigned to which subject by which *SOAs*. For each assignment, we can also specify other parameters like the delegation depth (0 means no delegation) and the time constraints[4] on the assignment.

For example, we can take in consideration a university environment with *"student"*, *"staff"* and *"professor"* as domains[5], and *"Student"*, *"Staff"* and *"Professor"* respectively as roles. The consequent Role Assignment Policy, with no delegation and no time restriction, can be written like the follow code:

[4]Policy time constraints, which are optional, over-rule any validity time in the attribute certificate.

[5]This means that we will have a SubjectPolicy like: <SubjectDomainSpec ID="student"> <Include LDAPDN="ou=student, o=PERMIS,c=gb"/ >< /SubjectDomainSpec> etc.

```
<RoleAssignmentPolicy>
        <RoleAssignment ID="RoleAssignment1">
            <SubjectDomain ID="student"/>
            <RoleList>
                <Role Type="permisRole" Value="Student"/>
            </RoleList>
            <Delegate/>
            <SOA ID="SOA"/>
            <Validity/>
        </RoleAssignment>
        <RoleAssignment ID="RoleAssignment2">
            <SubjectDomain ID="professor"/>
            <RoleList>
                <Role Type="permisRole" Value="Professor"/>
            </RoleList>
            <Delegate/>
            <SOA ID="SOA"/>
            <Validity/>
        </RoleAssignment>
         <RoleAssignment ID="RoleAssignment3">
            <SubjectDomain ID="staff"/>
            <RoleList>
                <Role Type="permisRole" Value="Staff"/>
            </RoleList>
            <Delegate/>
            <SOA ID="SOA"/>
            <Validity/>
        </RoleAssignment>
   </RoleAssignmentPolicy>
```

Figure 3.7: Role Assignment Policy Example

3.2.5 Target Policy

The Target Policy specifies the target domains managed by the policy. They are specified in the same way of the subject domains as LDAP DNs using Include and Exclude statements but they can also be HTTP-like URLs.

```
<TargetPolicy>
    <TargetDomainSpec ID="door">
        <Include LDAPDN="dc=kent,dc=ac,dc=uk">
          <Exclude LDAPDN="dc=library,dc=kent,dc=ac,dc=uk"/>
        <Include>
        <ObjectClass Name="Doors"/>
    </TargetDomainSpec>
    <TargetDomainSpec ID="unsecured">
        <Include URL="http://kent.ac.uk/"/>
    <TargetDomainSpec>
```

```
    <TargetDomainSpec ID="secured">
       <Include URL="https://kent.ac.uk/" />
    <TargetDomainSpec>
</TargetPolicy>
```

Figure 3.8: Target Policy Example

The union of all the domains specified in the TargetPolicy element is called the coverage domain of the policy. A requested target that is not a member of the coverage domain of the policy can never be mentioned in any of the access rules of that policy because this is checked at policy parsing time. Note that the opposite is not true: it is perfectly possible for a requested target to belong to the coverage domain of the policy but not have any access rule mention this particular target.

3.2.6 Action Policy

The actions that the users can perform are specified in the Action Policy. Every target supports different actions on it so, all of them have to be specified in the Action Policy as a list. Each action is defined with a name and a sequence of arguments.

Supposing to have the previous types of targets, we will have then actions for the door and actions for the link. However, it should be clear that in a case like this it doesn't make much sense to "open" on the target http://kent.ac.uk/, nor does it make much sense to "GET" on a door. So, we can restrict the applicability of the different actions by specifying a TargetDomain inside the relevant Action element. This is the recommended way of constructing policies and it is done as follows:

```
<ActionPolicy>
    <Action ID="open-door" Name="open">
                <TargetDomain ID="door" />
    </Action>
    <Action ID="close-door" Name="close">
                <TargetDomain ID="door" />
    </Action>
    <Action ID="get-data" Name="GET" />
                <TargetDomain ID="secured" />
                <TargetDomain ID="unsecured" />
```

```
    </Action>
    <Action ID="post-data" Name="POST" />
                <TargetDomain ID="secured" />
                <TargetDomain ID="unsecured" />
    </Action>
</ActionPolicy>
```

Figure 3.9: Action Policy Example

3.2.7 Target Access Policy

The last one and the most important part of the policy is the Target Access Policy. It is a set of Target Access clauses and it permits to make access control decisions. In fact, each target access clause grants an user with a specified set of role permissions to carry out the specified actions on the specified list of targets, but only if the conditions of the optional *IF* clause are true. Note that the policy works implicitly as Deny All and only specifying explicitly the permission as a rule in the target access clause, this can be granted.

Target Access specifies a set of roles and this means that a user must possess all these roles in order to gain the specific access/privilege.

After the set of roles, the Target List clause lists the targets taken from the domain. For each target are specified the granted actions[6].

Constraint *RBAC* is implemented in the policy with the *IF* clause. It specifies the conditions which must be satisfied in order for the action to be granted. A condition is comprises:

- a comparison (logical) operator,
- the *LHS*[7] operand, described by its source, name and type, and
- a series of one or more variables or constant values against which the LHS operand is to be compared.

The comparison operator belongs to a set of operators defined in the schema, such as EQ, GT, Substring, Subset etc.

[6]If no actions are specified then all actions that the target supports are granted.
[7]LHS is informal shorthand for the left-hand side of an equation.

The *LHS* operand is, in general, a Context ADI as defined in [ISOACF]. It is the context information available in the *AEF*[8] and it describes security-relevant properties of the context in which an access request occurs. In detail, it is implemented by *PERMIS* with the environmental variables. If it is strongly application dependent then the policy creator need to refer to the *AEF Reference Manual* for the list of the environment variables. An example of an application independent environmental variable is the time of day.

Constants are the *RHS* operands and they comprise types and values. They are compared against the *LHS* variables using the specified operator. The resulting value will be true or false.

The following example allows the student to open to the door only in limited gap of time (after 8 am and before 9 pm).

```
<TargetAccess>
  <RoleList>
    <Role Type="permisRole" Value="Student"/>
  </RoleList>
  <TargetList>
    <Target>
      <TargetDomain ID="door"/>
      <AllowedAction ID="open-door"/>
    </Target>
  </TargetList>
  <IF>
    <AND>
      <GE>
        <Environment Parameter="time" Type="Time"/>
        <Constant Type="Time" Value="*-*-*T08:00"/>
      </GE>
      <LE>
        <Environment Parameter="time" Type="Time"/>
        <Constant Type="Time" Value="*-*-*T21:00"/>
      </LE>
    </AND>
  </IF>
</TargetAccess>
```

Figure 3.10: Action Policy Example

[8] Access Control Enforcement Function. This is the same as the Policy Enforcement Point.

3.3 How write a new policy

In the previous section we described individually each part of the policy, explaining what each one is used for and how to write it. Now we are going to write a whole policy combining all the single parts together.

Writing a policy is a complex work of analysis and modeling that needs a deep study of the system. Therefore we need to start from a model of a real situation and then create a policy according to it. The model represents an abstract view of an environment for which the policy need to be written. It defines the subjects that interact in the system, the targets with which the subjects interact, the type of interactions and how they can or cannot interact each other. Only after an accurate analysis that permits to identify all the interesting characteristics of the system , we can formalize these characteristics in the policy with the formal chosen language. Note that if the policy is only on a specific aspect of the real case, such as the access the departments of a company, so the model will be only on the same aspect.

Let's consider a real case of a university, but it would be kept in mind that this model can be easily applied in other situations such as companies or hospital systems (obviously with some adjustments). Among the several matters of the university, the one we are interested in is access to the different facilities by the personnel.

Starting analysing the most significative personnel, different identities are identified: *Administrator*, *Administrative Staff member*, *Secretary*, *Professor*, *Researcher*, *MSc Student*, *BSc Student*, *Phd Student*. Each identity corresponds to a role in the university and consequently to a set of permissions. Obviously the three types of students have in common the property to be simply a student but each one adds different particularities. The subject domains are represented by the found identities. For each identities correspond its domain such as for Administrator the *Administrators domain* as well as the Professors and so on. As can be easily imagined, the found roles inside the university are a hierarchy and one example is in the Figure 3.11. The Administrators are at the top, the students are at the bottom and in the middle we have a hierarchy based on the privileges and the responsibility where the

professors have the researchers as children that in turn they have the students as children.

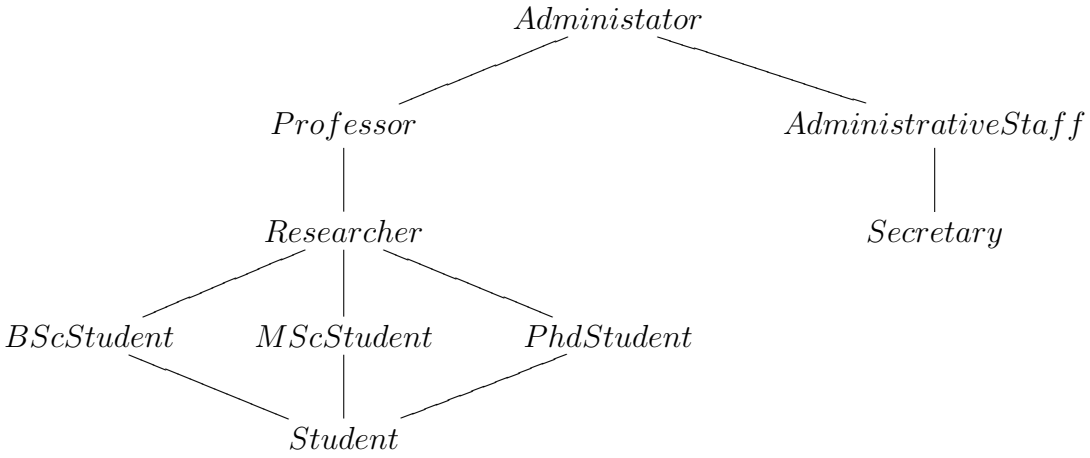

Figure 3.11: University Role Hierarchy

The facilities which have to be controlled are seen as resources. In the our university environment, we are interested in the *Library*, the *Laboratories*, the *Departments*, the *Research Offices* and *Administrative Offices*. Obviously in a university there are more of them that could be important but in the example we choose just these.

The only interaction that can make sense in our model is the action of *Access*. In fact, a subject can *access* to a facility.

Describing which interactions may or may not be made, could be quite complicated but, since using a *Deny All* policy, we are just interested to the interactions that CAN BE MADE. For example, students can access the library (but just between 8:00 and 23:00), the MSc Students can access the laboratories, the Researchers can access the research office, the professor to the department and the administrators can access the administrative offices. Everything else not in the policy and is not admitted. Using the inheritability that the Hierarchical *RBAC* provides, the professor can also access to the library as well as to the laboratory and so on. With all of these we have almost created a complete model that satisfies our purposes.

Whereupon, we make some consideration about: how the *ACs* are issued, the role's attributes, and also who is the *SOA*. In our case, the *ACs* are issued

by an internal *SOA*, called *USOA* (University Source Of Authority), that stores everything in a *LDAP* directory system.

We have defined most of the things, and what we need to do now is to write those things in the policy using the *XML* language and the schema as a guide. A possible policy that we could have as result is the following:

```
<X.509_PMI_RBAC_Policy OID="UniveristyPolicyExample">
<!-- The definition of the subject domains -->
    <SubjectPolicy>
        <SubjectDomainSpec ID="student">
            <Include LDAPDN="ou=student, o=Example, c=gb" />
        </SubjectDomainSpec>
        <SubjectDomainSpec ID="admin">
            <Include LDAPDN="ou=admin, o=Example, c=gb" />
        </SubjectDomainSpec>
        <SubjectDomainSpec ID="staff">
            <Include LDAPDN="ou=staff, o=Example, c=gb" />
        </SubjectDomainSpec>
    </SubjectPolicy>
<!-- The definition of the role hierarchy -->
    <RoleHierarchyPolicy>
<!-- uniRole is a role type invented right now so its OID does not
     have sense -->
        <RoleSpec OID="1.2.3.4.5.6.7.8.9" Type="uniRole">
<!-- The hierarchy is the same of the descriptions  -->
<!-- The Student role has no Subrole being it a leaf of the
    hierarchy tree -->
            <SupRole Value="Student" />
            <SupRole Value="Admin">
                <SubRole Value="Professor" />
                <SubRole Value="AdminStaff" />
            </SupRole>
            <SupRole Value="Researcher">
                <SubRole Value="BSc" />
                <SubRole Value="MSc" />
                <SubRole Value="PHd" />
            </SupRole>
            <SupRole Value="Professor">
                <SubRole Value="Researcher" />
            </SupRole>
            <SupRole Value="MSc">
                <SubRole Value="Student" />
            </SupRole>
            <SupRole Value="PHd">
                <SubRole Value="Student" />
            </SupRole>
            <SupRole Value="BSc">
                <SubRole Value="Student" />
```

```
            </SupRole>
            <SupRole Value="Secretary"/>
            <SupRole Value="AdminStaff">
                <SubRole Value="Secretary"/>
            </SupRole>
        </RoleSpec>
    </RoleHierarchyPolicy>
<!-- The definition of the SOAs -->
    <SOAPolicy>
<!-- Only one SOA is defined using just the internal one to issue
    the ACs -->
        <SOASpec ID="SOA" LDAPDN="cn=USOA, ou=admin, o=Example, c=
            gb"/>
    </SOAPolicy>
    <RoleAssignmentPolicy>
        <RoleAssignment ID="RoleAssignment1">
            <SubjectDomain ID="student"/>
            <RoleList>
                <Role Type="uniRole" Value="Student"/>
                <Role Type="uniRole" Value="MSc"/>
                <Role Type="uniRole" Value="PHd"/>
                <Role Type="uniRole" Value="BSc"/>
            </RoleList>
                                <!-- No delegation are defined -->
            <Delegate/>
            <SOA ID="SOA"/>
<!-- No validity restrictions are defined -->
            <Validity/>
        </RoleAssignment>
        <RoleAssignment ID="RoleAssignment2">
            <SubjectDomain ID="admin"/>
            <RoleList>
                <Role Type="uniRole" Value="Admin"/>
                <Role Type="uniRole" Value="AdminStaff"/>
            </RoleList>
            <Delegate/>
            <SOA ID="SOA"/>
            <Validity/>
        </RoleAssignment>
        <RoleAssignment ID="RoleAssignment3">
            <SubjectDomain ID="staff"/>
            <RoleList>
                <Role Type="uniRole" Value="Professor"/>
                <Role Type="uniRole" Value="Resercher"/>
                <Role Type="uniRole" Value="Secretary"/>
            </RoleList>
            <Delegate/>
            <SOA ID="SOA"/>
            <Validity/>
        </RoleAssignment>
    </RoleAssignmentPolicy>
```

```
<!-- The definition of the target domains -->
    <TargetPolicy>
        <TargetDomainSpec ID="Library">
            <Include LDAPDN="ou=library,o=Example,c=gb" />
        </TargetDomainSpec>
        <TargetDomainSpec ID="Laboratory">
            <Include LDAPDN="ou=laboratory,o=Example,c=gb" />
        </TargetDomainSpec>
        <TargetDomainSpec ID="Department">
            <Include LDAPDN="ou=department,o=Example,c=gb" />
        </TargetDomainSpec>
        <TargetDomainSpec ID="ResearchOffice">
            <Include LDAPDN="ou=resOffice,o=Example,c=gb" />
        </TargetDomainSpec>
        <TargetDomainSpec ID="AdministrativeOffice">
            <Include LDAPDN="ou=adminOffice,o=Example,c=gb" />
        </TargetDomainSpec>
    </TargetPolicy>
    <ActionPolicy>
        <Action ID="Access" Name="Access" />
    </ActionPolicy>
<!-- The definition of the target access rules -->
    <TargetAccessPolicy>
        <TargetAccess ID="TargetAccess1">
            <RoleList>
                <Role Type="uniRole" Value="Student" />
            </RoleList>
            <TargetList>
                <Target>
<!-- With the target is defined also the allowed actions -->
                    <TargetDomain ID="Library" />
                    <AllowedAction ID="Access" />
                </Target>
            </TargetList>
<!--The access restriction is specified using the invironment
    parameter "time" compared with a constant of the same type
    defined with the wanted time. -->
            <IF>
                    <AND>
                      <GE>
                        <Environment Parameter="time" Type="Time"/
                            >
                        <Constant Type="Time" Value="*-*-*T08:00" /
                            >
                      </GE>
                      <LE>
                        <Environment Parameter="time" Type="Time"/
                            >
                        <Constant Type="Time" Value="*-*-*T23:00" /
                            >
                      </LE>
```

```
                    </AND>
                </IF>
        </TargetAccess>
        <TargetAccess ID="TargetAccess2">
            <RoleList>
                <Role Type="uniRole" Value="MSc"/>
            </RoleList>
            <TargetList>
                <Target>
                    <TargetDomain ID="Laboratory"/>
                    <AllowedAction ID="Access"/>
                </Target>
            </TargetList>
        </TargetAccess>
        <TargetAccess ID="TargetAccess3">
            <RoleList>
                <Role Type="uniRole" Value="Research"/>
            </RoleList>
            <TargetList>
                <Target>
                    <TargetDomain ID="ResearchOffce"/>
                    <AllowedAction ID="Access"/>
                </Target>
            </TargetList>
        </TargetAccess>
        <TargetAccess ID="TargetAccess4">
            <RoleList>
                <Role Type="uniRole" Value="Professor"/>
            </RoleList>
            <TargetList>
                <Target>
                    <TargetDomain ID="Department"/>
                    <AllowedAction ID="Access"/>
                </Target>
            </TargetList>
        </TargetAccess>
        <TargetAccess ID="TargetAccess5">
            <RoleList>
                <Role Type="uniRole" Value="AdminStaff"/>
            </RoleList>
            <TargetList>
                <Target>
                    <TargetDomain ID="AdministrativeOffice"/>
                    <AllowedAction ID="Access"/>
                </Target>
            </TargetList>
        </TargetAccess>
    </TargetAccessPolicy>
</X.509_PMI_RBAC_Policy>
```

Figure 3.12: University Policy Example

This is an example of a policy that could be passed to *PERMIS* to make access decisions and that can work perfectly in a real situation. But this is not the only scope where the policy can be written. We can write the policy in the same way to protect either files in the filesystem or printers in a department or equipment in a factory. An interesting study case is the policy used to protect web resources; A more detailed explanation of how to write a web resource policy is given in Appendix A.

The policies are not written just for access controls, they are used by the Credential Validation Service and by the Delegation Issuing Service. The policies are exactly the same but what matters to them two is, in particular, the *RoleAssignmentPolicy*. With this particular part the *CVS* can check the trusted *SOAs* and which attributes these *SOAs* ares trusted to issue. The *DIS*, on the other hand, uses the policy to know which attributes can be delegated and with which restrictions.

3.4 Policy Editor

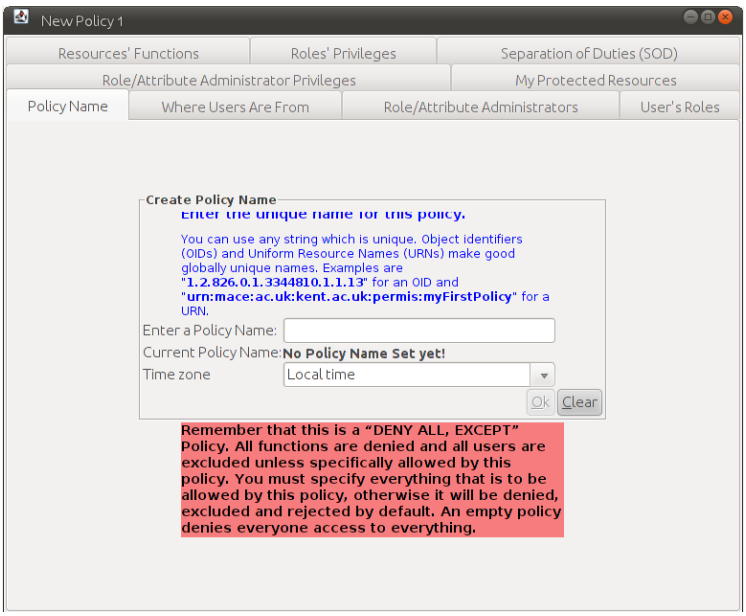

Figure 3.13: Policy Editor Screen: Policy Name

PERMIS provides some tools to help the policy administrator to write a

policy and to test it. One of these tools is the Policy Editor. It is a java application written around the *PERMIS* code. It helps to write a policy guiding the user through various steps. Of course it cannot replace the role of the modeling but, with a good model behind, it can help to write down the policy.

It provides a graphical interface with two modes to create a policy: the wizard and the normal mode. The wizard guides the user in a series of steps where the user is asked questions to produce the policy. In the normal mode, the user has a graphic tab for each policy part such as Subject Policy, Target Policy, Target Access Policy etc. We will take in consideration just the normal mode, as this is less intuitive.

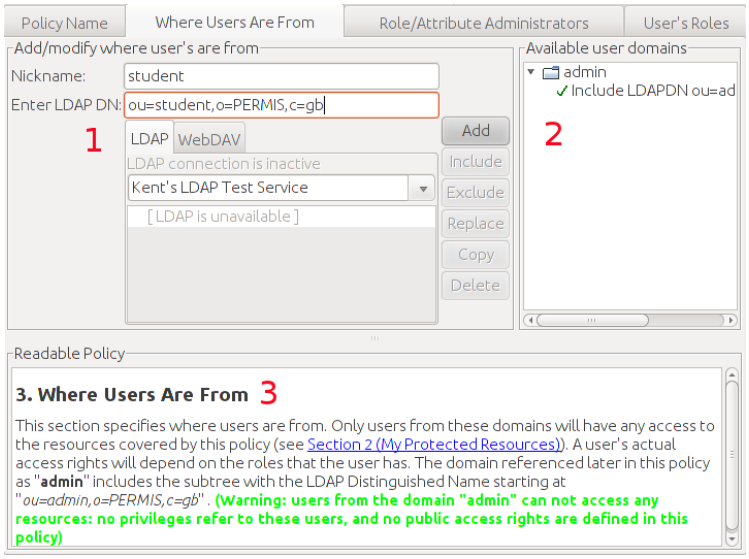

Figure 3.14: Policy Editor Screen: Subject Policy

First of all, the Policy Editor (*PE*) asks you to give a name for the policy, it will be the *OID* of the policy. The figure 3.13 shows the main screen when the user inserts the name of the policy, and the figure also shows the other tabs of the policy editor. In the red box is immediately specified that the policy is DENY ALL, and everything is defined is also permitted.

The figure 3.14 represents the Subject Policy tab, in (1) the user defines the subject domain and (2) lists the domains already defined. The Policy Editor can be connected to an *LDAP* server to retrieve the *LDAP DN* automatically,

without specifying it by hand. (3) provides a readable version of the policy, in this case it puts in to words the *admin* domain. The readable version is always present in each tab to help the user to understand better what he is writing.

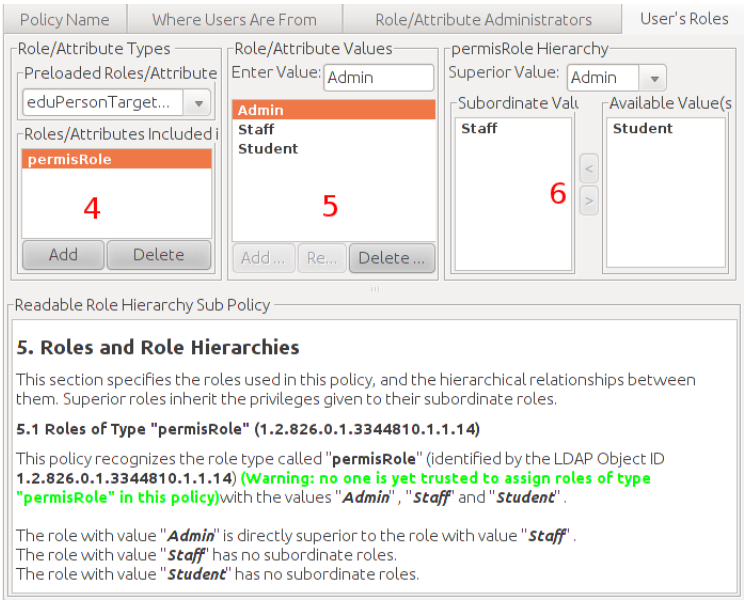

Figure 3.15: Policy Editor Screen: Role Hierarchy

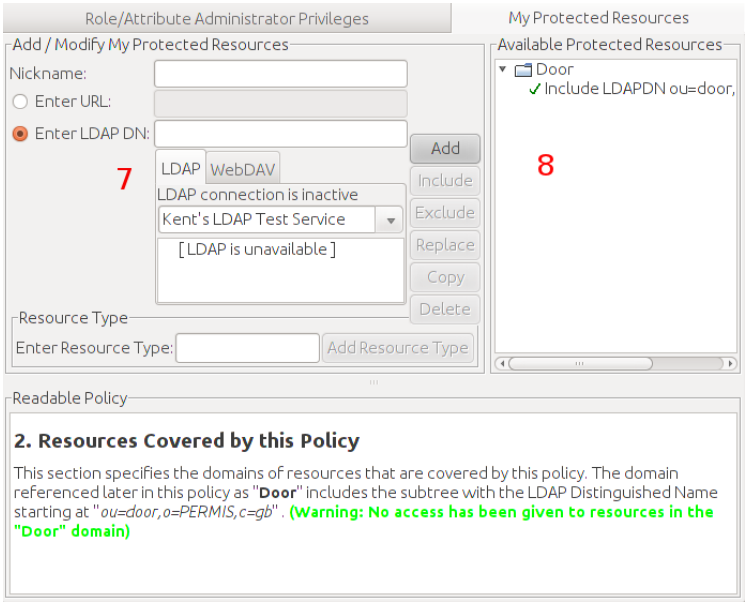

Figure 3.16: Policy Editor Screen: Target Policy

In the figures 3.15 3.16 we can see the role hierarchy definition and the Target Policy. In (4) is defined the type of the role, in (5) the name of the role and in (6) its place in the hierarchy. The Target Policy tab has the same structure as the Subject Policy tab: (7) defines new targets and (8) lists the targets already defined.

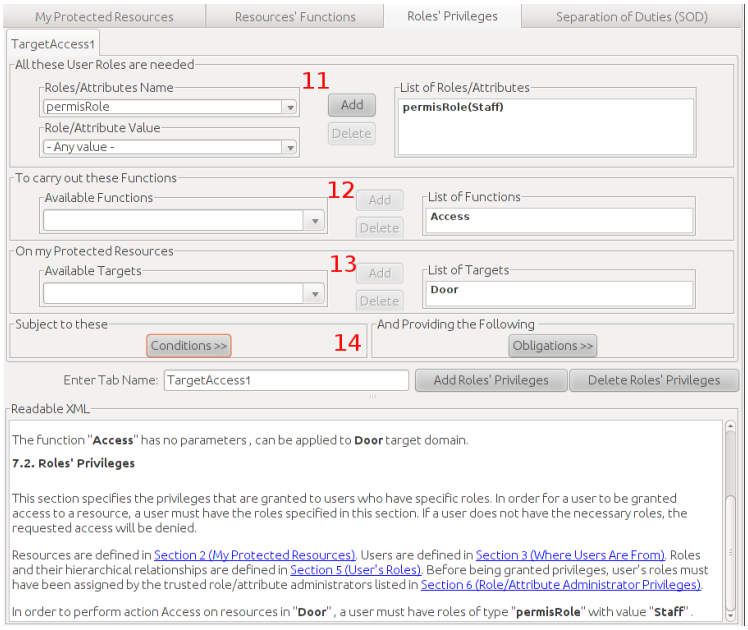

Figure 3.17: Policy Editor Screen: Target Access Policy

The Target Access Policy is created using the tab showed in the figure 3.17. This tab uses the information already inserted in the other tabs and creates target access policy combining these information. There are 3 lists where in (11) is the user chose the role, in (12) chose the action and (13) the target. This tab provide a button (14) to add conditions in the target access policy.

These condition are the same specified with the IF statement in the *XML* version of the policy. Of course more target access policies can be created by just clicking on the *add role's privileges* button, or removed by clicking on the *delete role's privileges*.

After going through all the tabs and completing all the steps we can examine the final version of the whole policy either in *XML* or in the readable version. The screen where the policy is printed is showed in the figure3.18.

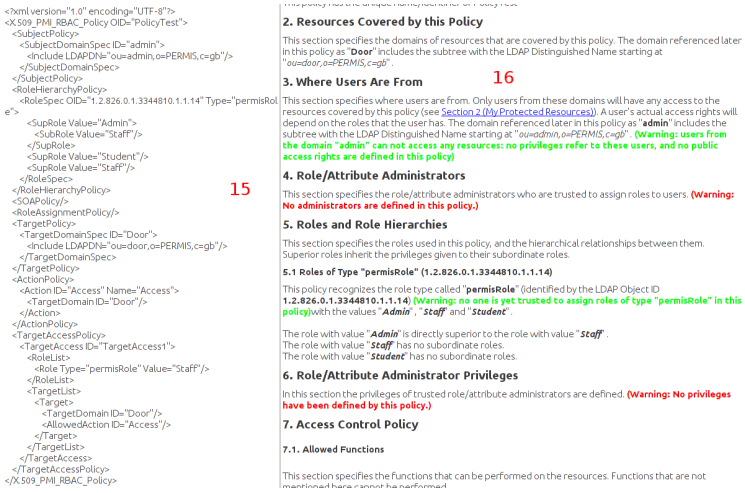

Figure 3.18: Policy Editor Screen: Policy view

3.5 Policy Tester

Once we have created the policy we cannot be completely sure that what we wrote respects the model and then that the policy does what we expect. Obviously we need a method to test the policy, to check if, when it will be applied in the real environment, it will work correctly. In fact, let's keep in mind that the policy is used to protect sensitive resources, so we cannot permit to compromise these resources due for a policy misconfiguration.

In order to test a policy and to be sure that the results will be the ones expected, *PERMIS* provides a software called Policy Tester. The Policy Tester as well as the Policy Editor is a java application written around the *PERMIS* code. This means that it uses the same methods and the same interfaces that *PERMIS* will use in reality.

It takes a policy and then prepares different use cases with known results with which to test the policy. If the policy tester, according with the policy, returns the aspected results then the policy is correct for those use cases. The correctness of the policy increases withe number of the use cases tested on it. Consider an example of use case a request such as *"a student wants access to the library"*.

The policy tester interface provides four different tabs to create use cases.

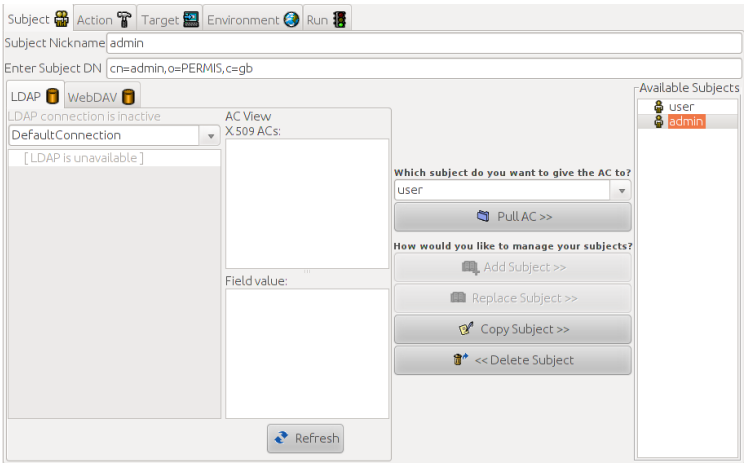

Figure 3.19: Policy Test Screen: Subjects Tab

The first one (Figure 3.19) the user can add a new subject providing the nickname and the *LDAP DN* in the same way of the policy editor. But adding this information is not enough. The policy tester, for the subject, needs also either an *AC* or a Role Text file. Clicking with the right button of the mouse on the added subject, we can add one of them. The role text file has a particular structure which specifies the role type, the holder of the role (the *LDAP DN* of the subject which is associated with the role text file), the *SOA* and the name of the role as described in the Listening 3.20. Each subject needs its own Role Text File or *AC*.

```
HOLDER=ou=student ,  o=PERMIS, c=gb
ISSUER=cn=SOA, ou=admin , o=PERMIS, c=gb
ROLE=permisRole
VALUE=Student
```

Figure 3.20: Role Text File Example

In the second tab, we add the actions that we want to test and in the next tab we add the targets. In the Environmental tab (Figure 3.21) we specify all the environmental variables but the types of these have to be defined in the configurations. The default types, already configured in the PT, are the time and the date.

The last tab is the most important one because it is the one that actually

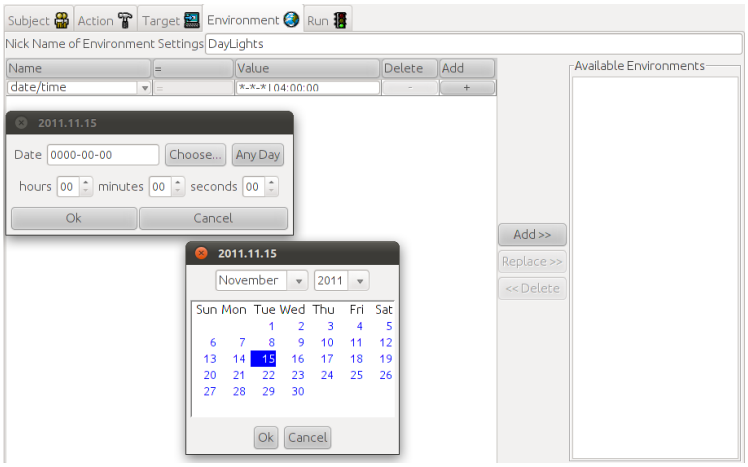

Figure 3.21: Policy Test Screen: Environment Tab

performs the tests. As showed in the Figure 3.22, the user selects the policy to test. The user then can select the subject, the action, the target and the environment variable for the use case and in the end run the test. The result of the test is printed in the section at the bottom, showing the response from the PDP as well as the information of the test. The Run tab provides also a button to perform all the tests in one go. The "Run All" button combines

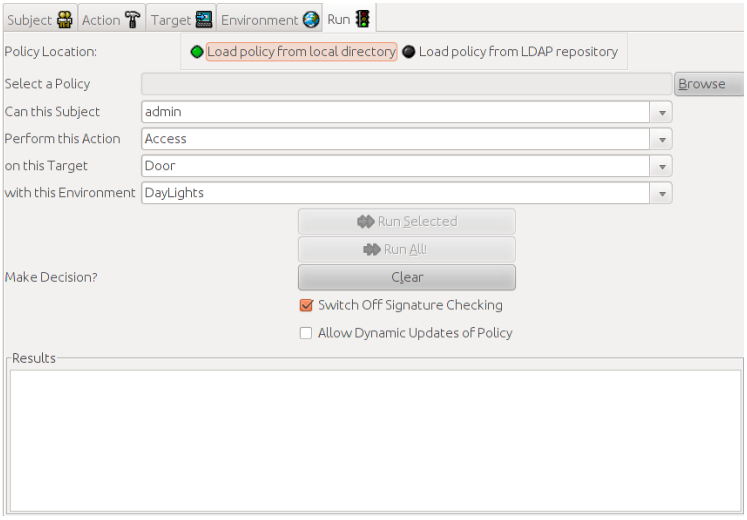

Figure 3.22: Policy Test Screen: Run Tab

all the subjects with all the actions with all the targets to create use cases

with all possible combinations of requests. The user will then check the result of each test to be sure that the policy provides correct responses even with unexpected combinations.

This last feature is really useful because it can permit to find unexpected behaviors of the policy with particular requests that the user would never have thought to test.

Chapter 4

Whitelist and Blacklist

We are now entering in the main part of the thesis, where we will present the actual research work that has been done. In this chapter, the whitelisting and blacklisting conceptual models will be presented and used to express our types of policies, in particular the exception rules. The rules will be written using a natural language formalism instead of using the *PERMIS* one, with the aim to provide an abstract interpretation of the rules.

4.1 Whitelist and Blacklist

Whitelist and Blacklist are lists of particular entries, whether *account name*, *email address*, *IP* etc. These lists can be expressed in very different ways and use very different languages. The main difference between the two is the way the system uses them. If the system reads a list of entries considering only those entries trusted, then that list is a whitelist. For the blacklist, it is the other way around. The list is of entries which are not trusted and everything else is trusted. This means that a single list can have two opposite meanings (white or black) just changing the way how the system interpreters it.

The system denies access to all by default, except the entities in the whitelist. This is useful if we have a few trusted entries and a lot of untrusted ones.

Whitelist benefits:

- protection from unknown identities

- fast decision making

Whitelist drawbacks:

- difficult to manage where there are a lot of exceptions from the norm
- time consuming to set up if there are a lot of trusted entities

Listing just the few things permitted, the system is protected from all the unknown threats and the security administrator does not have to take care of them. Unfortunately, it often happens that the permitted clauses are not easy to write especially if they include exceptions or special cases. Also, a whitelist policy may need a lot of study of the system in order to create an appropriate model of everyone who is trusted.

In comparison, the Blacklist is made of a list of untrusted entities which are not allowed access. The system denies access to everyone in the list. This is useful if the unsafe entities are known and any others are considered to be safe.

Blacklist benefits:

- protection from every known malicious identity
- easy to set up if only a few untrusted entities

Blacklist drawbacks:

- needs numerous updates
- lets unknown attackers have access
- can only be updated after an attacker is known, which may be too late

It is impossible to know all the malicious entities since you cannot enumerate infinity. There will always be things that you do not know and therefore cannot be included in the blacklist. This means that in a critical system it is not advisable to use a blacklist.

Particularly, the whitelisting and the blacklisting are used in *antivirus*, *IDS/IPS*, *spam filter* ex.
Blacklisting was the standard *de facto* method for these systems until recently. Now whitelisting has become the new solution to counteract the numerous updates and *0-day* attacks, which represent the main blacklisting problem. The priority problem is that the blacklist databases are becoming too big to be distributed or to be managed easily. In general, the trusted subjects are less then the untrusted ones, so the whitelist is easier to manage.

There is not one best choice between white or black list. It all depends on what you need. Probably a combination of the two is a good solution.

Example whitelist: the rule specifies who can access what.

> *"All students can access the Library, Computer Center and Dining Room."*
> *"All administrators can access the Administrative Office and Senate."*

```
...
    Student PERMIT ACCESS
                @Library
                @Computer_Center
                @Dining_Room
    Administrator PERMIT ACCESS
                @Administrative_Office
                @Senate
...
```

Example blacklist: the rule specifies who cannot access what.

> *"All students and staff cannot access the Administrative Office and Senate."*

```
...
    Student, Staff DENY ACCESS
                @Administrative_Office
                @Senate
...
```

4.2 Exception rules

We now start to talk about the exception rules that are the subject of this thesis.

To represent the exception rule we use the example of role hierarchy in Figure 4.1 of a University system in which there are the Admin role at the top of the graph and the Student role at the bottom. A superior role inherits all the privileges allocated to its subordinate roles.

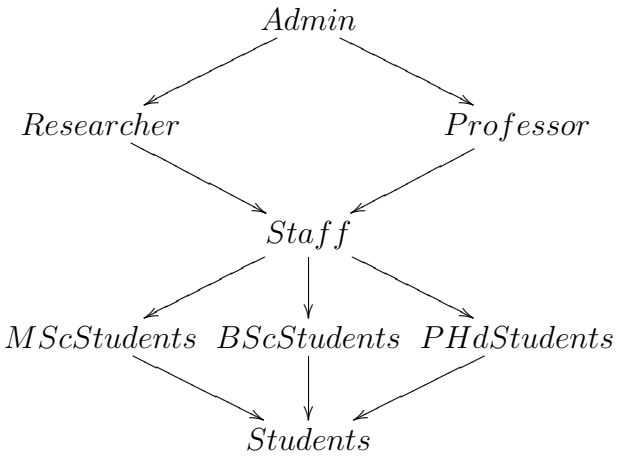

Figure 4.1: Example role hierarchy

As we already said, to write the rule we use the following components:

- **Subject** - this represents the user that executes the action. It belongs at the Subject Domain

- **Subject Domain** - this specifies the domain of the users who may be granted roles within the policy, only these may be authorized to access resources.

- **Target** - this represents the resource that is request by the subject.

- **Action** - this represents the action that the subject want to perform.

Along with the previous roles hierarchy, we use two *Subject Domain*: *Kent* and *Computing*.

With this model we can give some examples of different types of exception policies, which depend on the target role or the subject domain for the exception.

Exception from the domain of subject A set of permissions is given to a domain of subjects and a subset or a single subject can be excluded.

> *"Everyone from Kent can access the library except staff from Computing."*

The universal set, in this case is all the people of the University of Kent, and they can access the library, but the subset of the Computing Department staff cannot.

> *"Everyone from Kent can access to the library except Matteo."*

Or

> *"Everyone from Computing can access to the library except Matteo."*

In this case a single subject is excluded.

Exception for superior role When we give a privilege to a role any superior role has the same privilege but we can exclude one or more superiors.

> *"All staff can access the laboratory, Admin cannot."*

With this rule the staff and all its superior can access at the laboratory except the Admin.

"All Students can access the library except PHdStudents."

With this rule all students and all their superiors can access the laboratory except the PHd Students. Even if staff is superior of PHdStudent and Student, it inherits the union of the all subordinate roles, so the staff has permissions to access the library.

Exception for subordinate role This type of rule does not make much sense because if a privilege was given to a role, the subordinate role does not have it or inherit it, so we cannot give an exception rule to a role that already is excluded.

Exception for subject The target of the exception rule is not a role but a subject. Action is denied to a subject even if its role can perform it.

"All staff can access the laboratory except Matteo."

With this rule the staff and all its superiors roles can access the laboratory but Matteo cannot.

"All Students can access the library except Matteo."

All Students and all their superior roles can access the library but Matteo cannot. The difference between this rule and the previous domain rule is that in this one *Student* is a role while *Computing* is a subject domain in the other.

These types of rules are useful if the access must be denied to a particular subject or group of subjects.

Above we have seen all the possible examples of exceptions that can be made in our types of policies. But how can we represents these rules in *PERMIS* policies without violating the constraints of the actual policies and respecting the schema? This will be the content of the next chapter that will explain the solutions we found and we developed to permit writing these particular exception rules in *PERMIS*.

Chapter 5

Blacklist PERMIS Policies

In this chapter we will describe in detail the problems for the representation of exception rules and how we have solved it. More then one solution is presented and for each one we will underline the benefits and the drawbacks. Most of the work has been on an analysis of the problem and on researching the best way to solve it. In the end, the implementation of the solution was quite easy. The only big problem with the implementation phase was understanding the complexity of *PERMIS*.

5.1 Introduction

Coming back to the beginning, the aim of this work was to find a way to represent exception rules in *PERMIS*. In fact, *PERMIS* has a *PDP* (Policy Decision Point) that is a monotonic decision engine, in that everyone is denied access except for those that are allowed by the rules. This means that it uses whitelist policies where adding more rules grants more people access and there is no way of reducing access by adding more rules[1]. Its downside is that it can make it difficult to represent certain clauses with exception rules, e.g. "All students are granted access except MSc students".

There is not only one way to write such clauses in *PERMIS* but some of them cannot permit the maximum expressivity and others use mechanisms designed for different purposes. A good final solution has been developed and

[1]This is the big deal of respecting the monotony of the decision engine. Any deny rule violates the monotony.

it also extends the expressivity of the *PERMIS* policies in general. In the next sections these different solutions to the problem will be presented and some parts of the implementations will be shown.

5.2 Exception rule in PERMIS Policies

We now consider the examples of exception clauses from the previous chapter. We actually take those examples and we try to find a way to represent them changing the *PERMIS* code and the policy schema as little as possible. In fact, *PERMIS* has a really big code based and it is really complex in how it is structured and how it is written. The PDP is almost the core of *PERMIS* and so it is a very delicate part of the system. Every single little change could create big consequences in a lot of different parts of *PERMIS*, producing then undesirable behaviors.

Because it so delicate and sensitive, we started considering solutions that do not touch the core *PERMIS* code at all. Taking the first exception case of the previous chapter: ***Exception from the domain of subjects***. Considering the clause "Everyone from Kent can access to the library except staff from the Computing Department", this can be easily represented in the existing *PERMIS* policy. According with the schema, in the *subject domain* can be used the include/exclude statement. As we already saw, with the include we specify the *LDADN* of the subjects in the domain, but on the other hand, with the exclude statement we actually exclude the subjects from the domain. Obviously we can use the exclude statement to represent the exception clause in the policy. Figure 5.1 shows how the subject domain will appear for the specified clause.

```
<SubjectPolicy>
   <SubjectDomainSpec ID="ExceptionKent">
      <Include LDAPDN="dc=Kent,c=GB" />
      <Exclude LDAPDN="ou=computing,dc=Kent,dc=GB" />
   </SubjectDomainSpec>
</SubjectPolicy>
```

Figure 5.1: Example of subject domain with the exclude statement.

Different considerations come out from this example. Using an exclude statement in the domain means that all the times that the domain *Kent* is used, the *Computing Departement* will be always excluded. In fact, the subject domain is valid for all the access rules in the policy. If we want to use the *Kent* domain with the *Computing Department* included in another access rule of the same policy, this solution wont provide the correct results. If we specify a second domain with all the Kent personal without any exclusion statement, this will take precedence over the first domain, since the union of all domains is used by *PERMIS*. Hence the excluded subdomain will be ignored.

As we can see, this solution is not really flexible and also it is applicable only for the exclusion of the subject from the domain for all access rules. Indeed, if we have a more complicate rule we will need to adapt the policy to represent the exception rule.

Theoretically when this type of rule is used the *set subtraction* operation is needed, where the excluded subset is subtracted from the authorized set.

Let S set of students and B the subset of S of BSc Students:

$$\mathbf{B} \subset \mathbf{S} \tag{5.1}$$

then the rule "All student can access at the library except the BSc Student" can be:

$$\text{Library access permit to } \mathcal{S}\backslash\mathcal{B}. \tag{5.2}$$

Or

$$\forall x \in \mathcal{S}\backslash\mathcal{B} \qquad LibraryAccess(x). \tag{5.3}$$

Unfortunately, the set subtraction does not work at the moment because the *PERMIS* policy does not have *"set"* in the role representation and all the valid domains are merged together (set union).

Looking at the schema, became natural to say that we could use the policy statement *IF*. The *IF* statement specifies the conditions which must be satisfied

in order for action be performed and uses the Environmental Information to retrieve the information needed. Inside, the boolean operations (e.g. *AND*, *OR*, *NOT*) can be used.

We can start by defining a new environmental parameter called *subjectID* to represent the *ID* of the subject and then use this parameter when representing the rule *"All students can access the library, computer center and dining room but Matteo cannot enter to the library"*. In fact, the policy schema specifies that any attributes are allowed in the Environment clause. So the subject id or the subject name or any other attribute we need to represent the clause can be used. The rule could be:

```
...
<TargetAccess>
  <RoleList>
        <Role Type="permisRole" Value="Student"/>
  </RoleList>
  <TargetList>
        <Target>
                <TargetDomain ID="library-doors"/>
                <AllowedAction ID="open-door"/>
        </Target>
  </TargetList>
  <IF>
    <NOT>
      <EQ>
        <Environment Parameter="ID" Type="String"/>
        <Constant Type="String" Value="Matteo"/>
      </EQ>
    </NOT>
  </IF>
</TargetAccess>
<TargetAccess>
  <RoleList>
        <Role Type="permisRole" Value="Student"/>
  </RoleList>
  <TargetList>
        <Target>
                <TargetDomain ID="ComputerCenter-doors"/>
                <AllowedAction ID="open-door"/>
        </Target>
        <Target>
                <TargetDomain ID="DiningRoom-doors"/>
                <AllowedAction ID="open-door"/>
        </Target>
```

```
  </TargetList>
</TargetAccess>
...
```

Figure 5.2: Using IF statement to provide an Exception rule

This example explains a way to represent another case of the clause: ***Exception for subject***.

The rule *"All students are granted access at the library except MSc students"* is not that different from the previous case. In this rule we have the role MSc student, so we have to add the role to the possible Environmental parameters and use the same previous technique. The more the rules are complicated, the more complicated it is to represent them in *PERMIS*. For example, the rule *" All student can access the library and the bedroom. Matteo is a student but cannot access the library."* cannot be represented in only one *PERMIS* rule. The target accesses in the policy are grouped for subject and not for target, so it is necessary to group them by targets as required by *PERMIS*: one for the library target and one for the bedroom target. In this way the usual Environmental Parameters can be used without any change. As we can see, every small change in the natural language rules could produce very different *PERMIS* rules.

This solution uses the current *PERMIS* policy, but it is possible only if the *PEP* is modified in order that it passes the role twice to the *PDP* (once in the subject object and once in the environment) and it ensures the *CVS* first pulls all the subject's roles. This is acceptable if we have to use the present *PERMIS* policy.

But, using the Environment Parameter to represent the exception rule is not a clean solution. In fact, the way to bypass this rule is for a MSc student to only claim the student role and not the MSc role. A user could simply not send the negative attributes and would gain access. This is why negative attributes (such as MSc student) must be pulled by the *CVS*. We cannot trust the user to send them, because he might not do it. More then this, the main issue is that the Environment now has a completely different purpose. It should manage only the variables that concern the *PEP* and not the user.

So the *PEP* must call the *CVS* for it to pull the user attributes, then the *PEP* must copy the roles to the environment and subject fields, then call the *PDP*, and this is not a clean solution. A better solution is to alter the schema and allow conditions on roles without the *PEP* having to put the roles in the environment i.e. the code will pick up the roles from the subject object and uses them in the comparison. Unfortunately this creates several problems when respecting the *NIST RBAC* standard.

The correct writing of these rules is strictly dependent on the role hierarchy. It is therefore necessary to use a proper hierarchy of roles.

Suppose we now change the role hierarchy. Using the new hierarchy in Figure 5.3, we try to write again the exception rule *"All students are granted access to the library except MSc students"*.

This now introduces some ambiguity into the exception policy. Is the exception meant to be for any user who has the MSc Student role explicitly or implicity (by inheritance) or only for users who are explicitly MSc Students, and not for those who inherit its privileges such as Professors and Admin. We need to determine what the precise meaning of the exception rule should be before determining if the IF clause works correctly or not. There are thus two interpretations of this exception rule:

1. All students are granted access to the library except MSc students only

2. All students are granted access to the library except anyone who has the role of MSc student or above.

If the second meaning is intended, then the correct way to implement this would be to change the role hierarchy so that access to the library is given to Students, and MSc students are not made superior to Students.

We therefore conclude that this exception rule should apply to MSc students only and not to its superior roles and that the first meaning is the correct one.

In fact, using the first semantics *IF* statement works correctly, because the inheritance of the permissions is not effected. The behavior we aspect is

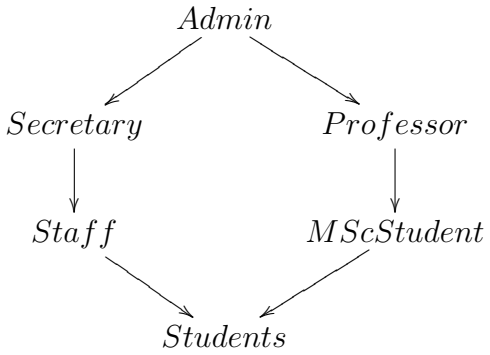

Figure 5.3: Example of role hierarchy 2

that, if the student role has a permission say, delete file, all the superior roles inherit the same permission. But if the MScStudent role is excluded one of its permissions in the IF statement, say access the library, this does not stop the superior roles from still inheriting this permission, and they are not excluded by the IF statement. So the Professor will still have the permissions of the Student and can enter the library. NOTE that in theory, the Admin should have the student permissions, inheriting them from Secretary and Staff and therefore should not be affected by the IF statement.

This same ambiguity will appear with the combination of the two policies but it will be explained in detail in the next section.

To test all these new solutions we used the *Policy Tester* software. We wrote different use cases and we wrote a couple of policies to represent different scenarios. We applied these use cases and we received the expected behavior from the *Policy Tester*. We also set up a *PERMIS* standalone server with our policies and we made specific *XACML* and *SOAP* requests to it using a software tool called *soapUI*[SOAPUI]. *soapUI* is a free and open source Functional Testing solution. With an easy-to-use graphical interface, *soapUI* allows to create and execute automated functional, regression, compliance, and load tests. In our case, it makes easy to work with *SOAP* and *REST*-based Web services.

The solutions we found so far are correct but they can cover just a small set of cases, where the rules are simple. For example, the policy "All student can access the library except researchers and all their superiors" cannot be easily

represented with the present *PERMIS* policy. To have a mechanism that could be used to write every type of rule, adding then expressivity to the policy, it is necessary to create a *Grant All PDP* and combine it with the already existing *Deny All PDP*.

5.3 Grant-all PERMIS decision engine

After a series of tests and tries we realized that the modifications and the adaptations of the policies are not enough to represent all the exception rules. The rules we can express just using the existing policy are just a few. We decided then to make enhancements and updates to the *PERMIS PDP*.

However, *PERMIS* implements the *PDP* not just as a single element but instead with a complex system of classes woven together. The parser classes are a part of them.

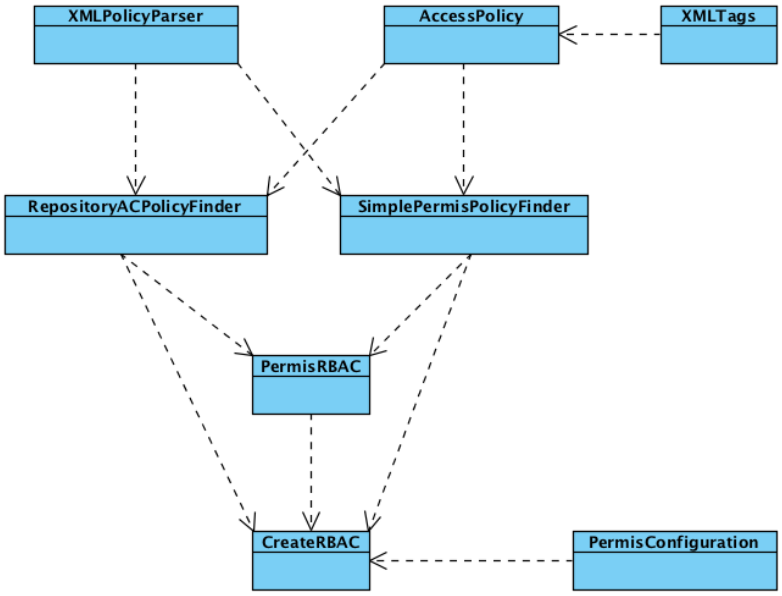

Figure 5.4: UML Classes Schema

First of all, we have selected just the classes that concern our purpose, and then we have understood how they interact with others. Figure 5.4 represents the class schema. The interesting classes are the following:

- **CreateRBAC** : this class is used to create a *PermisRBAC* object according to the *PermisConfiguration* object. It creates the correct *policy finder* according to the type of the policy specified in the configuration.
- **PermisConfiguration** : contains all the information specified in the configuration file. When the configuration file is read, this class is instantiated and the attributes of the object are filled with all the information.
- **PermisRBAC** : this class represents the shell of the PDP. It is the class that has the method *getCreds* that makes the credential validation and it has the method *authzDecision* that takes the decisions. It uses the policy finder passed by parameter.
- **SimplePermisPolicyFinder** : This class implements the *PolicyFinder* interface, and it permits to load a policy from a plain text *XML* file. It creates the *XMLPolicyParser* to parse the policy and the *AccessPolicy*, the core of the *PDP*.
- **SimplePermisACPolicyFinder** : This is a simple Policy Finder that can be instantiated from an instance of an *X.509* Attribute Certificate. It has almost the same function of the *SimplePermisPolicyFinder* but it uses an *X.509* instead of a simple *XML* file.
- **XMLPolicyParser** : is the parser for the *XML* policy.
- **AccessPolicy** : This is the class representing the Target Access Policy. It delivers the decision on whether a user with a certain set of credentials is allowed to access a target. It is the core of the PDP and it makes the decisions.
- **XMLTags** : This class contains the names of the *XML* tags the *XMLParser* looks for.

Once this subset of classes were selected it became easy to find the work flow of the decision making procedure and so understand how this works. This work flow can be summarised in the following steps: the *CreateRBAC* creates the

PermisRBAC, the *getCreds* method of the latter is used to validate credentials and the *authzDecision* is used to make the decision; the policy finder is passed to the *PermisRBAC* and it manages the parsing of the policy according to its type; the *authzDecision* method checks the credentials of the subject and calls the policy finder method *getAccessPolicy* to retrieve the *AccessPolicy* object created by the parser; Once the *authzDecision* method has the *AccessPolicy* object, it calls its method *responce* passing the credentials, action, and target to make the decision; In the end the response is returned.

So what we need is to go through this work flow and make the needed changes.

First of all, we start from the policy. It is necessary that the policy has something to specify that it is a blacklist policy. In this way, the parser identifies the right type of policy and so it knows the way to manage it. We decided then to add in the first statement of the policy the attribute *DenyBased*. When this attribute is not present, the policy is a whitelist. Figure 5.5 shows the use of this attribute.

```
<X.509_PMI_RBAC_Policy OID="PolicyTestBlacklist" DenyBased="true">
```

Figure 5.5: Use of the new DenyBased attribute in the policy.

After this change, the policy can remain the same, as what is important is the meaning of it. The only other change that we made to the policy is adding a new statement called *DeniedAction* to take the place of the *AllowedAction* one. The reason is primarily cosmetic. In fact, being a blacklist policy it does not make sense to have an *AllowedAction* statement when the meaning of it is to deny that action. So we decided to require the use of that statement in place of the AllowedAction one in the blacklist policy. An example of the use of it is in Figure 5.6.

```
        <TargetAccess ID="TargetAccess2">
                <RoleList>
                        <Role Type="permisRole" Value="MSc"/>
                </RoleList>
                <TargetList>
                        <Target>
```

```
                                <TargetDomain ID="Library"/>
                                <DeniedAction ID="Access"/>
                        </Target>
                </TargetList>
        </TargetAccess>
```

Figure 5.6: Use of the new DeniedAction statement in the policy.

Once the policy has been designed, we need to modify the *PERMIS* code to accept this new policy and to change the way it makes decisions.

First of all the parser has to parse the new statements for the blacklist policy. It is also necessary that the statement *DeniedAction* is not present in a whitelist policy or the *AllowedAction* is not present in a blacklist policy. These checks have to be made by the parser.

After that, the decision making part of *PERMIS* has to be modified to manage the blacklist policies. A lot of little changes have been done along the code but the big one is in the response. Figure 5.7 shows an effective piece of code for the responses provided by the AccessPolicy.

```
    /* Note of the access rules have granted access.
 */
if (exceptionSoFar == null) {
  // no exceptions occurred, so it's a Deny
    if (denyBasedPdp){
            return new PERMISResponse(true);
    }else {
            return new PERMISResponse(false);
    }
} else {
  // propagate the exception recorded
  throw exceptionSoFar;
}
```

Figure 5.7: Modified Code for the response in the AccessPolicy file.

All the other changes concern the use of the PermisRBAC and the policy finder. It is important to see that after the changes for the blacklist concept are present in the code, the PDP knows that the new type of policy exists and it knows how to make decisions according to it.

The new *PDP*, in the end, is the same as before but now it works with two different types of policies. It recognises the policy type , it parses it correctly

and it makes the right decisions. The table of the current decisions made in relation with the previous ones are shown in the table 5.1. It provides the decision when a Target Access Rule (TAR) that grants the request is "Present", "NOT Present" or "Not Sure" in the policy.

	TAR Present	TAR NOT Present	TAR Not Sure
WhiteList PDP	GRANT	DENY	NOT APPLICABLE or INDETERMINATE
Blacklist PDP	DENY	GRANT	NOT APPLICABLE or INDETERMINATE

Table 5.1: Expected responses for a specific request.

Not Sure means that the policy does not have enough information to resolve the request so the *PDP* returns NOT APPLICABLE or INDETERMINATE accordingly with what is missing.

But this is not the end. In this way we just created a reversed *PDP* that can manage the blacklist but this it is not enough for our purposes. Using just this new *PDP* we can express negative rules but not our exception rules. To reach this point, it is necessary to combine both of the two *PDPs* in one single *PDP*, that acts like a single one that uses two policies.

5.4 Combination of grant-all and deny-all PERMIS decision engine

From having two separate deny all and grant all *PDPs* to have a single combined *PDP* the step is small. It is just enough to create a wrapping *PDP* around the two single *PDPs* that provides the same interface, being in this way completely transparent for those who use it. The only problem is to identify exactly where to create the wrapper, at which level.

The *PermisRBAC* that represent effectively the starting point of the *PDP* takes in reality a policy finder as a parameter and this is linked to a single policy, so it cannot be wrapped around by a combination class. It is almost the same for the policy finder. It parses the policy and so it works with a

single policy. At this point, it is too complicated to rewrite one of these two class and also it is out of the scope of the current dissertation to make large changes to *PERMIS*.

Changing the point of view, we can see that *PERMIS* uses mainly two types of interfaces: *XACML* and *SAML*. These two interfaces handle the requests respectively, and their classes take the *PermisRBAC* object to make decisions. Particularly the *SAML* handler class is created starting from a *XACML* handler class and the latter takes as parameter the *PermisRBAC* with the linked policy.

Accordingly, it follows the creation of a combination class of *XACML* handlers that takes two different *PermisRBAC* with their policies. Inside this combination class, the single *XACML* handlers are created with the correct *PermisRBAC* objects. The class has to have the same structure and type of the *XACML* handler class (*Xacmlv2Handler.java*) so it implements the interface class *XamlHandler*. We called our class *CombinationXmlHandler*. After that, we mainly overrode the method *handleRequest* that is the main method and the one that returns the decision for the specific request. In order to return the final response, the new combination entity needs to combine all the possible combinations of responses of the handlers. Figure 5.2 shows the responses combinations.

Whitelist	Blacklist	Combination
-	DENY	DENY
-	INDETERMINATE	INDETERMINATE
DENY	GRANT or NOT APPLICABLE	DENY
PERMIT	GRANT or NOT APPLICABLE	PERMIT
INDETERMINATE	GRANT or NOT APPLICABLE	INDETERMINATE
NOT APPLICABLE	GRANT or NOT APPLICABLE	NOT APPLICABLE

- means All possible responses.

Table 5.2: Table of the results of the combination of PDPs responses.

This table shows how we decided to manage the combination of the responses. First of all, it is clear that we gave priority to the blacklist *PDP*, in fact only if the blacklist one returns NOT APPLICABLE or GRANT the control is passed to the whitelist one. In an access control environment and particularly when we are protecting a resource, for security matters is better to

deny an access than permit the access to an unauthorised entity. The response valuation rule is cerated respecting this concept. The INDETERMINATE response from the blacklist *PDP* is returned as final response because this means that an error is occurred during the processing of the request or something is missing then for the same reason of before is better to return immediately the problem to the caller and leave to this the duty to handle it. The NOT APPLICABLE response instead has not the same behavior. Differently from the single blacklist *PDP*, here the *PDPs* are combined so the policies are combined too, meaning that if some domain is missing in the blacklist policy it is maybe present in the whitelist one and so the handling of the request is passed to the whitelist *PDP* that will provide the final response.

As we already introduced, there is another thing that works differently from what we expected. In Section 5.2, we presented the role hierarchy in Figure 5.3 and we mentioned the fact that the inheritance of privileges using exception rules. However, when we use our blacklist PDP the inheritance of exceptions rules is enabled, so that even if theoretically the Admin role should have the inherited student privileges, in our implementation this is no longer true. The following example explains this issue fully.

Using the same hierarchy in Figure 5.3, say we want express, using the combined *PDP*, the exception rule "Students can access to the library except MSc Students".

We have to suppose also that both of the two policies define exactly the same role hierarchy. At this point, if *PERMIS* receives a request from an Admin to access the library, according to the policy and the combined *PDP*, the response is deny. This is in contrast with what we said theoretically should happen, but we consider this correct for the following reasons. The first one is that if there is a conflict between the two *PDPs* we give the blacklist one highest priority, for the same reason we said before about security matters: is better to deny then permit if we want to protect a resource. The second reason is a bit more sophisticated and is about the attributes (or roles) and privileges. If an attribute (role) is in a hierarchy, it contains inside itself all the children attributes (roles). A subject that claim a particular attribute can obviously

```
...
<TargetAccess>
 <RoleList>
  <Role Type="permisRole"
      Value="Student"/>
 </RoleList>
 <TargetList>
  <Target>
   <TargetDomain ID="Library"
       />
   <AllowedAction ID="access"
       />
  </Target>
 </TargetList>
</TargetAccess>
...
```

Figure 5.8: TargetAccessPolicy in the Whitelist Policy

```
...
<TargetAccess>
 <RoleList>
  <Role Type="permisRole"
      Value="MScStudent"/>
 </RoleList>
 <TargetList>
  <Target>
        <TargetDomain ID="
            Library"/>
        <DeniedAction ID="
            access"/>
  </Target>
 </TargetList>
</TargetAccess>
...
```

Figure 5.9: TargetAccessPolicy in the Blacklist Policy

Figure 5.10: The two policies handled by the combined PDP.

claim any children attributes. Accordingly, it follows that to gain the access the subject has just to claim an attribute (role) lower in the hierarchy. In the specific case, the subject can make the request with the attribute *staff* and then *PERMIS* will grant the access. But if a professor requests access then he will inherit the MSc student role and therefore be denied access. The solution to this problem is to change the role hierarchy in the GrantAll *PDP*, so that no roles are superior to the MSc student role, and then no roles will inherit the exception to be denied access to the library.

Chapter 6

Validation

In this part we describe how the system was validated to prove that the system is working correctly. Some validation tests will be shown as well as the corresponding responses. We also describe the performance test we made for the new combined *PDP*.

6.1 Correctness Tests

In the previous chapter all the found solutions have been presented and discussed, now is necessary to describe the ways we used to prove their correctness.

When the exception rules were expressed using just the Exclude/Include statements, it was not necessary to write particular tests. To test these we simply used the *Policy Tester* software. We wrote some use cases and a couple of policies to test and we submitted them to the PolicyTest. Finally, we compared the results of the PolicyTester with the expected ones to check if everything was right. The results were as expected.

Once we introduced the use of the *IF* statements to express exception rules, we ran our tests with the *soapUI*. This is because it is not possible to use the Policy Tester for these tests as it does not correctly create the environmental attributes. The *soapUI* is a testing software that permits to create easily *SAML* and *XACML* requests and then receive the respective responses. So we prepared different requests to represent different situations and then checked that *PERMIS* behaved correctly by the response returned. Of course to use

soapUI to make requests, a *PERMIS* standalone server needed to be set up.

Figures 6.1 and 6.2 show an example of an XACML request and the corresponding response from *PERMIS* printed by the *soapUI*.

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/
    envelope/" xmlns="urn:oasis:names:tc:xacml:2.0
    :context:schema:os">
<soapenv:Header/>
<soapenv:Body>
<Request>
<Subject>
<Attribute AttributeId="urn:oid:1.2.826.0.1.3344810.1.1.14"
    DataType="http://www.w3.org/2001/XMLSchema#string">
<AttributeValue>MSc</AttributeValue></Attribute>
</Subject>
<Resource>
<Attribute AttributeId="urn:oasis:names:tc:xacml:1.0
    :resource:resource-id" DataType="http://www.w3.org/2001/
    XMLSchema#string">
<AttributeValue>ou=lib-door,o=PERMIS,c=gb</AttributeValue>
</Attribute>
</Resource>
<Action>
<Attribute AttributeId="urn:oasis:names:tc:xacml:1.0:action:action
    -id" DataType="http://www.w3.org/2001/XMLSchema#string">
<AttributeValue>Access</AttributeValue>
</Attribute>
</Action>
<Environment>
<Attribute AttributeId="ID" DataType="http://www.w3.org/2001/
    XMLSchema#string">
<AttributeValue>Matteo</AttributeValue>
</Attribute>
</Environment>
</Request>
</soapenv:Body>
</soapenv:Envelope>
```

Figure 6.1: XACML Request in soapUI

```
<soapenv:Envelope xmlns:soapenv="␣http://schemas.xmlsoap.org/␣soap
    /envelope/">
<soapenv:Body>
<urn:Response xmlns:urn="urn:oasis:names:tc:xacml:2.0
    :context:schema:os">
<urn:Result>
<urn:Decision>Permit</urn:Decision>
<urn:Status>
<urn:StatusCode Value="urn:oasis:names:tc:xacml:1.0:status:ok"/>
```

```
</urn:Status>
</urn:Result>
</urn:Response>
</soapenv:Body>
</soapenv:Envelope>
```

Figure 6.2: XACML response in soapUI

Finally, it is necessary to test the last and more interesting but complex solution we developed: the combined *PDP*. This one is plugged in the *PERMIS* standalone server so we tested some examples of requests using *soapUI*.

All the *PERMIS* code is tested using *junit* so we chose to use this for our combined *PDP* as well. junit runs the tests for the PERMIS code all together, and if some tests fail, it means that our changes produce unexpected behaviours in other parts of the code.

Using this type of test for our code , it will produce test cases for future developments. Hence, we wrote some java test classes.

We prepared some regression tests using *junit* first of all to test if the parsing process is correct and then if with particular requests *PERMIS* provides the expected responses. In the first case, the parser should thrown an exception if there are any problems in the policy including the wrong use of the new statement *DeniedAction*. Figure 6.3 shows the code that checks this in the *junit* test class.

```
        try {
              engine = (new CreateRBAC(config)).
                 getPermisRBAC();
            } catch (PolicyParsingException e) {
              assertTrue(false);
            }
```

Figure 6.3: *junit* code for the checking of a correct parsing.

Of course this code is just a piece of testing code in one of the testing class.

To test the corrrectness of the responses, we prepared different hand made requests and then we submitted them to a standalone PDP set up with a blacklist policy. Combining together legal and illegal requests with the PDP,

we tested the responses and compared them with the expected ones. Figure 6.4 is an example of a testing method for a specific request with a specific policy where the request should be denied.

```
/**
 * Test blacklist.
 * Tests whether passing a correct blacklist
    policy and a proper request,
 * the engine recognises the policy correctly and
    makes a right decision.
 */
@Test
public void testDeniedAccess() {
  PERMISConfiguration config =
     getBaseConfiguration(true);
  createEngine(config);

  try {
    Subject sub = getSubject2();
    assertFalse(engine.authzDecision(sub, ACTION,
       TARGET, null).isAuthorised());
  } catch (PbaException e) {
    assertTrue(false);
  } catch (RFC2253ParsingException e) {
    assertTrue(false);
  }
}
```

Figure 6.4: *junit* code that checks a response of the DPD with an unauthorised request.

Running these tests, we have an idea of the behaviour of the system and the changes we made, and then if the behaviour is as expected. If a test goes wrong and it fails, it means that something in the process did not work as expected. We made a good analysis of the problem and a good planning of the solutions and this has permitted us to ensure that we have no fails in the tests. Of course, in this way there is no assurance of 100% correctness, but adding more tests will only show that the software can be considered more correct.

For our purposes, we are satisfied with the tests we made so far and with their results.

6.2 Performance Test

In every project, it is important to know which are the benefits of any new extensions, but also what are the costs of providing them. In our case, what we could lose is performance. The new combined PDP could spend too much time to make decisions and its benefits could be not enough compared to the handicap of a longer time.

It was necessary to have an idea of the times of the *PDPs*, so we created performance tests to calculate these times. The tests aimed to calculate the time that the *PDP* requires to resolve a request. We used a Benchmark testing framework called *BB* made by the Elliptic Group. It used a normal Whitelist policy and a normal Blacklist policy and then they were used together. The two policies are in Appendix B.

The results of the performance tests are shown in table 6.1[1].

	Mean	CI Deltas	Confidence Level
Deny All PDP	$128.480\mu\ s$	$-1.841\mu\ s, +1.680\mu\ s$	0.95
Grant All PDP	$128.985\mu\ s$	$-1.661\mu\ s, +1.787\mu\ s$	0.95
Combined PDP	$131.097\mu\ s$	$-1.800\mu\ s, +1.495\mu\ s$	0.95
Combined PDP [1]	$130.231\mu\ s$	$-1.700\mu\ s, +1.544\mu\ s$	0.95

[1] Combined PDP with an empty blacklist policy. The policy has no *TargetAccessPolicy* so it denies nothing. NOTE that this type of policy is used only for this test as it is not compliant with the schema that requires the *TargetAccessPolicy* statement.

Table 6.1: Table of the results of the performance tests.

The CI Deltas indicate the reliability of the mean. It is an interval calculated from the performance of each test, in principle different from sample to sample, that frequently includes mean. How frequently the observed interval contains the mean is determined by the confidence level.

As we can see from the results, we have good performances for both the Grant All PDP that and the Deny All PDP. They use exactly the same policy and they have almost the same performance. The Combined PDP has a really

[1]The tests are preformed on local system. The system has: Intel(R) Core(TM)2 Duo CPU E6750 @ 2.66GHz with 4096 KB of cache and 4 GB of RAM.

good performance. There is only a slightly delay between the time of the Combined PDP and the simple Deny All PDP and the reason is because most of the time is spent to elaborate the request and the context and only a little time is used to make the real decision.

With the good performance tests we conclude that the real gain of the new feature is up to expectations.

Chapter 7

Conclusion

In this last chapter we review the results that were achieved with the project and the related works. We also provide an overview of the possible future developments.

7.1 Achievements

According to the requirements of the project, the work was focussed on a new extension to express the exception rules, but it has involved more then just the *PERMIS* software but other entities of the infrastructure as well. Even if it is not strictly a part of *PERMIS*, the *PolicyTester*, for example, has been modified in order to help the tests of the new *PERMIS* functionalities for the exception rules. The new *PolicyTester* features permit us to make tests using the two types of policies by them self or combined together.

The real gain of the *PERMIS* infrastructure is the new feature of the PDP. The extension for the exception rules brought new interesting and useful abilities to the PDP. The enhanced *PERMIS PDP* extends the *PERMIS* expressivity, providing to the user the freedom to use either type of policy (Blacklist or Whitelist) according to its needs. It is not dependent only on the whitelist policies any more, but it can manage blacklists as well. This permits us to use *PERMIS* in different environments where the whitelist policy is unsuitable.

Ultimately, the combined PDP concludes the extension of the expressivity of the policies. It permits to use the whitelist and the blacklist policy at the

same time combining together the meaning of the two and providing a complete coverage from the threats. Even if before we had the same security from the threats, now we can have the same level providing a simple way to express it. In fact, combining the two policies, their writing process has a very lower complexity. Moreover, of course, this new feature achieves our main target: permit to write exception rules in the policy.

All of this has been achieved without losing anything on performance. The new features, in fact, do not add any significant additional time during the access control process.

7.2 Future Developments

The results that the project reached are good enough for the expressivity of *PERMIS* policies. What han not been done so far, is the integration of these new Grant All and Combined PDPs in all the extra features of *PERMIS*. In fact, the new PDPs contain only the basic *PERMIS* requirements. The BTG (Break The Glass) rules, Obligations as well as other extensions of the RBAC model are not fully integrated with the our new features, accordingly they are not supported in our model.

Possible future developments could be to increase the integration of the new PDPs with all the other extensions that *PERMIS* has.

It could be also interesting to introduce the concept of exception rules in the Policy Editor, especially in the part where a rule in natural language is translated in a PERMIS rule.

Appendix A

PERMIS Policies for Web Based Resources

PERMIS policies for web based resources are not different from any other *PERMIS* policy. However, the web resource policy has to include the specifics of the web-server protocol used by the browser (i.e. *HTTP*) and the specific way that web resources are named. This effects the way how targets and actions are specified in the policy.

A.1 Specifying Resources

Web servers refer their resources with URLs. The policy writer has to identify these URLs and the related subdirectories that may have access restrictions. Once he has done that, he has to group the resources in domains that have the same restrictions. As a general principle, domains with stricter access controls should be subordinate to domains with more relaxed access controls.

When we write the policy, we include these domains in the policy and for each of them the related *URLs*. For example, we can specify three domains: *public domain*, *restricted domain* and *top secret domain*. The public domain includes the *URL* www.mysite.com that is the public page of the site. The restricted domain includes the subdirectory www.mysite.com/restricted that is the administration directory of the site. In the end, the top secret domain includes the *URL* www.mysite.com/topsecret.php.

A.2 Specifying Actions

The actions are defined at the level of the web-browser to web server interaction protocol, *HTTP*. There are 8 actions that directly correspond to the *HTTP* methods, defined in the *RFC 2616*.

When the web server is configured to use *PERMIS*, for each request it interrogates the *PDP* in order to determinate whether the request is allowed or denied. Hence, the web server will pass the requested URL as target and the requested *HTTP* method name as action.

Don't forget that if an action is not mentioned in the *PERMIS* policy, it will be denied to everyone.

A.3 Set up *PERMIS* policies for less secure subordinate directories

As it stands, secure directories are subordinate to less secure directories but, some times, could be necessary restrict access to a superior directory. In this way, the exclude rule denies the access to a superior directory even if the one of subdirectory can be accessed.

Using the previous example, if in the *Restricted Domain* we have a particular resource to protect from who can access to *Top Secret* we should exclude these from the *Restricted Domain*.

Based on *PERMIS* policy schema, the *Exclude* and *Include* clause can be used to exclude or include directory from a domain.

In the definition of the Target Domain policy, for each target domain, we specify all the permitted resources with the *Include* clause and all the resources not permitted with the *Exclude* clause.

A.3.1 Exclude the access to subdirectory

As we have seen, the directory with stricter access control is subordinated to directory with more relaxed access control. In the policy, this is represented

including only the permitted directory in the domain and exclude the subdirctory with denied access.

For example, if we have this role:

"Everyone can access http://www.mysite.com/ but not http://www.mysite.com/restricted/ or http://www.mysite.com/cgi-bin/"
"Staff can access http://www.mysite.com/restricted/ but not http://www.mysite.com/restricted/secret"

the *PERMIS* policy could be:

```
...
<TargetPolicy>
        <TargetDomainSpec ID="PublicDomain">
        <Include URL="http://www.mysite.com/" />
                <Exclude URL="http://www.mysite.com/restricted" />
                <Exclude URL="http://www.mysite.com/restricted" />
        </TargetDomainSpec>
        <TargetDomainSpec ID="RestrictedDomain">
                <Include URL="http://www.mysite.com/restricted" />
                <Exclude URL="http://www.mysite.com/restricted/
                    secret" />
        </TargetDomainSpec>
</TargetPolicy>
...
```

A.3.2 Exclude the access to superior directory

In the same way, exclude the access to superior directory can be possible using *Include*/*Exclude* clause. The only difference is that all superior directory must be excluded.

For example, if we have this role:

"Everyone can access http://www.mysite.com/TopSecret/restricted/public but not http://www.mysite.com/TopSecret/restricted or http://www.mysite.com/TopSecret"
"Staff can access http://www.mysite.com/TopSecret/restricted but not http://www.mysite.com/TopSecret "
"Admin can access http://www.mysite.com/TopSecret "

the *PERMIS* policy could be:

```
...
<TargetPolicy>
        <TargetDomainSpec ID="PublicDomain">
        <Include URL="http://www.mysite.com/TopSecret/restricted/
            public"/>
                <Exclude URL="http://www.mysite.com/TopSecret/
                    restricted"/>
                <Exclude URL="http://www.mysite.com/TopSecret"/>
        </TargetDomainSpec>
        <TargetDomainSpec ID="RestrictedDomain">
                <Include URL="http://www.mysite.com/TopSecret/
                    restricted"/>
                <Exclude URL="http://www.mysite.com/TopSecret"/>
        </TargetDomainSpec>
        <TargetDomainSpec ID="TopSecretDomain">
                <Include URL="http://www.mysite.com/TopSecret"/>
        </TargetDomainSpec>
</TargetPolicy>
...
```

Using the combination of *Include/Exclude* clause in the *PERMIS* policy, the access control of the web based directory can be possible.

Then we have to use all the techniques we described in the thesis to represent the exception rules, so that we can write in *PERMIS* rules like:

"Student can access http://www.mysite.com/gallery/fairytale"
"BSc Student except Matteo can access http://www.mysite.com/gallery/"

Or,

"Staff except Researchers can access
http://www.mysite.com/projects/admin"

A complete description of this topic is provided in the paper *"How to Specify PERMIS Policies for Controlling Access to Web Based Resources"* available on the *PERMIS's* site.

Appendix B

Policies for the Benchmark tests

These two policies were used for the benchmark tests.

```
<?xml version="1.0" encoding="UTF-8"?>
<X.509_PMI_RBAC_Policy OID="PolicyTestWhitelist">
    <SubjectPolicy>
        <SubjectDomainSpec ID="student">
            <Include LDAPDN="ou=student, o=PERMIS, c=gb" />
            <Include LDAPDN="cn=dis, ou=admin, o=PERMIS, c=gb" />
        </SubjectDomainSpec>
        <SubjectDomainSpec ID="admin">
            <Include LDAPDN="ou=admin, o=PERMIS, c=gb" />
        </SubjectDomainSpec>
        <SubjectDomainSpec ID="staff">
            <Include LDAPDN="ou=staff, o=PERMIS, c=gb" />
            <Include LDAPDN="cn=dis, ou=admin, o=PERMIS, c=gb" />
        </SubjectDomainSpec>
    </SubjectPolicy>
    <RoleHierarchyPolicy>
        <RoleSpec OID="1.2.826.0.1.3344810.1.1.14" Type="
            permisRole">
            <SupRole Value="Student" />
            <SupRole Value="Admin">
                <SubRole Value="Professor" />
                <SubRole Value="Research" />
            </SupRole>
            <SupRole Value="Staff">
                <SubRole Value="BSc" />
                <SubRole Value="MSc" />
                <SubRole Value="PHd" />
            </SupRole>
            <SupRole Value="Professor">
                <SubRole Value="Staff" />
            </SupRole>
            <SupRole Value="MSc">
                <SubRole Value="Student" />
            </SupRole>
```

```
            <SupRole Value="PHd">
                <SubRole Value="Student"/>
            </SupRole>
            <SupRole Value="BSc">
                <SubRole Value="Student"/>
            </SupRole>
            <SupRole Value="Research">
                <SubRole Value="Staff"/>
            </SupRole>
        </RoleSpec>
    </RoleHierarchyPolicy>
    <SOAPolicy>
        <SOASpec ID="SOA" LDAPDN="cn=SOA, ou=admin, o=PERMIS, c=gb
            "/>
    </SOAPolicy>
    <RoleAssignmentPolicy>
        <RoleAssignment ID="RoleAssignment1">
            <SubjectDomain ID="student"/>
            <RoleList>
                <Role Type="permisRole" Value="Student"/>
            </RoleList>
            <Delegate/>
            <SOA ID="SOA"/>
            <Validity/>
        </RoleAssignment>
        <RoleAssignment ID="RoleAssignment2">
            <SubjectDomain ID="admin"/>
            <RoleList>
                <Role Type="permisRole" Value="Admin"/>
            </RoleList>
            <Delegate/>
            <SOA ID="SOA"/>
            <Validity/>
        </RoleAssignment>
        <RoleAssignment ID="RoleAssignment3">
            <SubjectDomain ID="staff"/>
            <RoleList>
                <Role Type="permisRole" Value="Staff"/>
            </RoleList>
            <Delegate/>
            <SOA ID="SOA"/>
            <Validity/>
        </RoleAssignment>
    </RoleAssignmentPolicy>
    <TargetPolicy>
        <TargetDomainSpec ID="Door">
            <Include LDAPDN="ou=lab-door, o=PERMIS, c=gb"/>
            <Include LDAPDN="ou=lib-door, o=PERMIS, c=gb"/>
            <Include LDAPDN="ou=class-door, o=PERMIS, c=gb"/>
        </TargetDomainSpec>
        <TargetDomainSpec ID="Library">
```

```
            <Include LDAPDN="ou=lib-door,o=PERMIS,c=gb" />
        </TargetDomainSpec>
        <TargetDomainSpec ID="Laboratory">
            <Include LDAPDN="ou=lab-door,o=PERMIS,c=gb" />
        </TargetDomainSpec>
        <TargetDomainSpec ID="Class">
            <Include LDAPDN="ou=class-door,o=PERMIS,c=gb" />
        </TargetDomainSpec>
    </TargetPolicy>
    <ActionPolicy>
        <Action ID="Access" Name="Access" />
    </ActionPolicy>
    <TargetAccessPolicy>
        <TargetAccess ID="TargetAccess1">
            <RoleList>
                <Role Type="permisRole" Value="Student" />
            </RoleList>
            <TargetList>
                <Target>
                    <TargetDomain ID="Door" />
                    <AllowedAction ID="Access" />
                </Target>
            </TargetList>
            <IF>
                <AND>
                    <OR>
                        <NOT>
                            <EQ>
                                <Environment Parameter="ID" Type="
                                    String" />
                                <Constant Type="String" Value="
                                    Matteo" />
                            </EQ>
                        </NOT>
                    </OR>
                </AND>
            </IF>
        </TargetAccess>
        <TargetAccess ID="TargetAccess2">
            <RoleList>
                <Role Type="permisRole" Value="Student" />
            </RoleList>
            <TargetList>
                <Target>
                    <TargetDomain ID="Library" />
                    <AllowedAction ID="Access" />
                </Target>
            </TargetList>
        </TargetAccess>
        <TargetAccess ID="TargetAccess3">
            <RoleList>
```

```
            <Role Type="permisRole" Value="Staff"/>
        </RoleList>
        <TargetList>
            <Target>
                <TargetDomain ID="Laboratory"/>
                <AllowedAction ID="Access"/>
            </Target>
        </TargetList>
    </TargetAccess>
    <TargetAccess ID="TargetAccess4">
        <RoleList>
            <Role Type="permisRole" Value="Student"/>
        </RoleList>
        <TargetList>
            <Target>
                <TargetDomain ID="Class"/>
                <AllowedAction ID="Access"/>
            </Target>
        </TargetList>
    </TargetAccess>
</TargetAccessPolicy>
</X.509_PMI_RBAC_Policy>
```

Figure 2.1: Whitelist Policy

```
<?xml version="1.0" encoding="UTF-8"?>
<X.509_PMI_RBAC_Policy OID="PolicyTestBlacklist" DenyBased="true">
    <SubjectPolicy>
        <SubjectDomainSpec ID="student">
            <Include LDAPDN="ou=student, o=PERMIS, c=gb"/>
            <Include LDAPDN="cn=dis, ou=admin, o=PERMIS, c=gb"/>
        </SubjectDomainSpec>
        <SubjectDomainSpec ID="admin">
            <Include LDAPDN="ou=admin, o=PERMIS, c=gb"/>
        </SubjectDomainSpec>
        <SubjectDomainSpec ID="staff">
            <Include LDAPDN="ou=staff, o=PERMIS, c=gb"/>
            <Include LDAPDN="cn=dis, ou=admin, o=PERMIS, c=gb"/>
        </SubjectDomainSpec>
    </SubjectPolicy>
    <RoleHierarchyPolicy>
        <RoleSpec OID="1.2.826.0.1.3344810.1.1.14" Type="
            permisRole">
            <SupRole Value="Student"/>
            <SupRole Value="Admin">
                <SubRole Value="Professor"/>
                <SubRole Value="Research"/>
            </SupRole>
            <SupRole Value="Staff">
                <SubRole Value="BSc"/>
                <SubRole Value="MSc"/>
                <SubRole Value="PHd"/>
```

```
            </SupRole>
            <SupRole Value="Professor">
                <SubRole Value="Staff"/>
            </SupRole>
            <SupRole Value="MSc">
                <SubRole Value="Student"/>
            </SupRole>
            <SupRole Value="PHd">
                <SubRole Value="Student"/>
            </SupRole>
            <SupRole Value="BSc">
                <SubRole Value="Student"/>
            </SupRole>
            <SupRole Value="Research">
                <SubRole Value="Staff"/>
            </SupRole>
        </RoleSpec>
    </RoleHierarchyPolicy>
    <SOAPolicy>
        <SOASpec ID="SOA" LDAPDN="cn=SOA, ou=admin, o=PERMIS, c=gb
            "/>
    </SOAPolicy>
    <RoleAssignmentPolicy>
        <RoleAssignment ID="RoleAssignment1">
            <SubjectDomain ID="student"/>
            <RoleList>
                <Role Type="permisRole" Value="Student"/>
            </RoleList>
            <Delegate/>
            <SOA ID="SOA"/>
            <Validity/>
        </RoleAssignment>
        <RoleAssignment ID="RoleAssignment2">
            <SubjectDomain ID="admin"/>
            <RoleList>
                <Role Type="permisRole" Value="Admin"/>
            </RoleList>
            <Delegate/>
            <SOA ID="SOA"/>
            <Validity/>
        </RoleAssignment>
        <RoleAssignment ID="RoleAssignment3">
            <SubjectDomain ID="staff"/>
            <RoleList>
                <Role Type="permisRole" Value="Staff"/>
            </RoleList>
            <Delegate/>
            <SOA ID="SOA"/>
            <Validity/>
        </RoleAssignment>
    </RoleAssignmentPolicy>
```

```
<TargetPolicy>
    <TargetDomainSpec ID="Door">
        <Include LDAPDN="ou=lab-door,o=PERMIS,c=gb" />
        <Include LDAPDN="ou=lib-door,o=PERMIS,c=gb" />
        <Include LDAPDN="ou=class-door,o=PERMIS,c=gb" />
    </TargetDomainSpec>
    <TargetDomainSpec ID="Library">
        <Include LDAPDN="ou=lib-door,o=PERMIS,c=gb" />
    </TargetDomainSpec>
    <TargetDomainSpec ID="Laboratory">
        <Include LDAPDN="ou=lab-door,o=PERMIS,c=gb" />
    </TargetDomainSpec>
    <TargetDomainSpec ID="Class">
        <Include LDAPDN="ou=class-door,o=PERMIS,c=gb" />
    </TargetDomainSpec>
</TargetPolicy>
<ActionPolicy>
    <Action ID="Access" Name="Access" />
</ActionPolicy>
<TargetAccessPolicy>
    <TargetAccess ID="TargetAccess1">
        <RoleList>
            <Role Type="permisRole" Value="Student" />
        </RoleList>
        <TargetList>
            <Target>
                <TargetDomain ID="Door" />
                <DeniedAction ID="Access" />
            </Target>
        </TargetList>
        <IF>
            <AND>
                <OR>
                    <NOT>
                        <EQ>
                            <Environment Parameter="ID" Type="
                                String" />
                            <Constant Type="String" Value="
                                Matteo" />
                        </EQ>
                    </NOT>
                </OR>
            </AND>
        </IF>
    </TargetAccess>
    <TargetAccess ID="TargetAccess2">
        <RoleList>
            <Role Type="permisRole" Value="Student" />
        </RoleList>
        <TargetList>
            <Target>
```

```
                    <TargetDomain ID="Library"/>
                    <DeniedAction ID="Access"/>
                </Target>
            </TargetList>
        </TargetAccess>
        <TargetAccess ID="TargetAccess3">
            <RoleList>
                <Role Type="permisRole" Value="Staff"/>
            </RoleList>
            <TargetList>
                <Target>
                    <TargetDomain ID="Laboratory"/>
                    <DeniedAction ID="Access"/>
                </Target>
            </TargetList>
        </TargetAccess>
        <TargetAccess ID="TargetAccess4">
            <RoleList>
                <Role Type="permisRole" Value="Student"/>
            </RoleList>
            <TargetList>
                <Target>
                    <TargetDomain ID="Class"/>
                    <DeniedAction ID="Access"/>
                </Target>
            </TargetList>
        </TargetAccess>
    </TargetAccessPolicy>
</X.509_PMI_RBAC_Policy>
```

Figure 2.2: Whitelist Policy

Bibliography

[ACW] *Access Control Models.* www.wikipedia.com.

[X509] *X.509.* www.wikipedia.com.

[LDAP] Timothy A Howes *An X.500 and LDAP Database: Design and Implementation.*

[SAMLW] *SAML.* www.wikipedia.com.

[SUNXACML] *Sun's XACML Implementation.* http://sunxacml.sourceforge.net/.

[IBMXACML] Manish Verma, IBM *XML Security: Control information access with XACML.* http://www.ibm.com/developerworks/xml/library/x-xacml/.

[OASISXACML] OASIS, *eXtensible Access Control Markup Language (XACML) Version 1.1* .

[NISTRBAC] Ravi Sandhu (LIST), David Ferraiola and Richard Kuhn (NIST) *The NIST Model For Role-Based Access Control: Towards A Unified Standard.*

[RBAC] Ravi Sandhu (LIST), Edward J. Coyane, Hal L. Feinstein and Charles E. Youman (1995) *Role-Based Access Control Models.* IEEE Computer, Volume 29, Number 2, February 1996, pages 38-47.

[ABAC] ITU-T Rec X.812 (1995) — ISO/IEC 10181-3:1996 *Security Frameworks for open systems: Access Control framework.*

[SCHEMA] MAKOTO MURATA Makoto Murata (IBM Tokyo Research Lab), Dongwon Lee (Penn State University), Murali Mani (Worcester Polytechnic Institute) and Kohsuke Kawaguchi (Sun Microsystems) Taxonomy of XML Schema Languages using Formal Language Theory ACM Journal Name, Vol. V, No. N, November 2004.

[ISOACF] ISO 10181-3 Access Control Framework. Information Technologies - Open System Interconnection.

[SOAPUI] soapUI. The Swiss-Army Knife of Testing. http://www.soapui.org/.