SEVENTH FRAMEWORK PROGRAMME
Challenge 1
Information and Communication Technologies

Trusted Architecture for Securely Shared Services

Document type: D7.1

Title:	Design of Identity Management, Authentication and Authorization Infrastructure

Work Package: WP7

Deliverable Number: D7.1

Editor: David Chadwick, University of Kent

Dissemination Level: Public

Preparation Date: 20 May 2010

Version: 2.1.1

Legal Notice
All information included in this document is subject to change without notice. The Members of the TAS³ Consortium make no warranty of any kind with regard to this document, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. The Members of the TAS³ Consortium shall not be held liable for errors contained herein or direct, indirect, special, incidental or consequential damages in connection with the furnishing, performance, or use of this material.

The TAS³ Consortium

Nr	Participant name	Country	Participant short name	Participant role
1	Katholieke Universiteit Leuven	BE	KUL	Coordinator
2	Synergetics nv/sa	BE	SYN	Partner
3	University of Kent	UK	KENT	Partner
4	University of Karlsruhe	DE	KARL	Partner
5	Technical University Eindhoven	NL	TU/e	Partner
6	Consiglio Nazionale delle Ricerche	IT	CNR	Partner
7	University of Koblenz-Landau	DE	UNIKOLD	Partner
8	Vrije Universiteit Brussel	BE	VUB	Partner
9	University of Zaragoza	ES	UNIZAR	Partner
10	University of Nottingham	UK	NOT	Partner
11	SAP research	DE	SAP	S&T Coordinator
12	Eifel asbl	FR	EIF	Partner
13	Intalio Ltd	FR	INT	Partner
14	Risaris Ltd	IR	RIS	Partner
15	Kenteq	BE	KETQ	Partner
16	Oracle	UK	ORACLE	Partner
17	Custodix nv/sa	BE	CUS	Partner
18	Medisoft bv	NL	MEDI	Partner
19	Symlabs SA	PT	SYM	Partner

Contributors

	Name	Organisation
1	David Chadwick, Lei Lei Shi, Stijn Lievens, Ana Ferreira, Kaniz Fatema	University of Kent
2	Sampo Kellomaki	Symlabs
3	Danny de Cock	KU Leuven
4	Marc Santos	University of Koblenz-Landau

Table of Contents

EXECUTIVE SUMMARY 6

1 INTRODUCTION 7

1.1 RESEARCH CONTRIBUTIONS 8
1.2 DOCUMENT STRUCTURE 9

2 OVERALL ARCHITECTURAL DESIGN OF THE IDENTITY MANAGEMENT, AUTHENTICATION AND AUTHORIZATION (IDMAA) INFRASTRUCTURE 10

2.1 FEDERATED IDENTITY MANAGEMENT 10
2.1.1 Identities, Identifiers and Attributes 10
2.2 AUTHORITATIVE SOURCES 11
2.3 AUTHENTICATION IN FIM 12
2.4 AUTHORIZATION IN FIM 13
2.5 THE OBLIGATION ENFORCEMENT INFRASTRUCTURE 19
2.5.1 The Obligation Service Interfaces 22
2.5.2 The Obligations Service 25

3 BREAK THE GLASS INFRASTRUCTURE 27

3.1 THE BTGI STATE VARIABLE 30
3.1.1 Resetting the BTGi State Variables 31
3.2 IMPLEMENTING BTG 32
3.2.1 Encoding the Permission to BTG response 35
3.3 ADAPTIVE AUDIT CONTROLS 35
3.3.1 Implementing Adaptive Adaptive Audit Controls 36

4 ATTRIBUTE AGGREGATION INFRASTRUCTURE 38

4.1 CONCEPTUAL MODEL 38
4.1.1 Link Registration Phase 38
4.1.2 Level of Assurance 40
4.1.3 Link Release Policy 42
4.1.4 Service Provision Phase 42
4.1.5 Using the LoA in Service Provision 45
4.2 MAPPING THE CONCEPTUAL MODEL TO STANDARD PROTOCOLS 46
4.2.1 Link Registration Protocol 46
4.2.2 Service Provision Protocols 47

5 MULTIPLE POLICY AUTHORIZATION EVALUATION INFRASTRUCTURE 50

5.1 THE MASTER PDP 50
5.2 STICKY POLICY CONTENTS 54
5.3 CONFLICT RESOLUTION POLICY 56

6 DYNAMIC DELEGATION OF CREDENTIALS INFRASTRUCTURE 58

6.1 REQUIREMENTS FOR WEB SERVICES DELEGATION OF AUTHORITY 58
6.2 DESIGN OF A DELEGATION OF AUTHORITY WEB SERVICE 60
6.3 IMPLEMENTING THE DELEGATION OF AUTHORITY WEB SERVICE 64
6.4 DESIGN OF A CREDENTIAL VALIDATION SERVICE 66
6.4.1 The Trust Model 67
6.4.2 The Credential Validation Policy 68
6.4.3 The CVS functional components 70
6.5 IMPLEMENTING THE INITIAL CVS 72
6.5.1 Delegation Tree Navigation 75

7 DYNAMIC MANAGEMENT OF POLICIES INFRASTRUCTURE 78

7.1 UPDATING THE COLLABORATION AUTHORISATION POLICY 83

8 INFRASTRUCTURE FOR THE DISTRIBUTED ENFORCEMENT OF STICKY POLICIES 85

8.1 THE APPLICATION PROTOCOL ENHANCEMENT (APE) MODEL 86
8.2 THE ENCAPSULATING SECURITY LAYER (ESL) MODEL 89
8.3 THE BACK CHANNEL MODEL 90
8.4 FUNCTIONALITY OF THE PEP 92
8.5 TRUST NEGOTIATION 93

9 EVENT HANDLING INFRASTRUCTURE & ITS APPLICATION TO ADAPTIVE AUDIT CONTROLS 95

10 AUTHORIZATION ONTOLOGY 96

10.1 OVERVIEW 96
10.2 THE AUTHORISATION ONTOLOGY MODEL 98

11 TRUST AND PRIVACY NEGOTIATION 110

11.1 INTRODUCTION 110
11.2 TAS3 TRUST NEGOTIATION CONCEPTS 112
11.2.1 Example negotiation where client reveals all required credentials 115
11.2.2 Example negotiation where client does not reveal all required credentials 115
11.3 CUP ACCESS CONTROL POLICIES AND TRUST NEGOTIATION 116
11.3.1 The TrustBuilder2 framework 117
11.3.2 Extending the TB2 framework 121
11.3.3 Future research directions 125

12 INTEGRATION OF CUP TRUST NEGOTIATION INTO THE PROTOTYPE........126
12.1 INTERNAL INTEGRATION 126
12.1.1 Service discovery.. 126
12.1.2 Service invocation 126
12.1.3 Policy enforcement.. 127
12.2 GUI INTEGRATION 127
12.2.1 Policy Editor... 128
1.1.3 Credential Editor 130
13 CONCLUSIONS 132
14 GLOSSARY 135
15 REFERENCES..138
APPENDIX 1. THE OBLIGATION SCHEMA AND JAVA INTERFACE 141
APPENDIX 2 THE CVS POLICY SCHEMA 148
APPENDIX 3. A REMOTE OBLIGATION ENFORCEMENT SCENARIO....................158

Executive Summary

This document describes the design of the identity management, authentication and authorization infrastructure, which is needed in order to achieve the security, trust and privacy objectives of the TAS^3 project.

Section 2 of this document describes the overall architecture of the identity management, authentication and authorization infrastructure. Section 2 also describes the obligation infrastructure that supports policy enforcement through the automatic execution of obligations (where this is possible). Section 3 describes the design of the Break the Glass (BTG) infrastructure. BTG allows users who are not normally authorized to access resources, to gain access after first "breaking the glass" in the full knowledge that they will have to answer later to management about this. Section 3 also describes how adaptive audit controls can be supported in order to support BTG policies. Both of these features are enabled through the obligation infrastructure described in Section 2. Section 4 describes the design of a credential aggregation infrastructure where user credentials can be retrieved, aggregated and validated in dynamically changing environments, even when the user is known by different identities at different identity providers. Section 5 describes the multiple policy authorization evaluation infrastructure, which will provide support for multiple authorization policies written in different languages to be evaluated and any conflicts between them to be resolved before the user is granted access to a resource. Section 6 describes the design of the infrastructure for the dynamic delegation of credentials between the various actors of the system, and the verification of these credentials using a Credential Validation Service. Section 7 builds on section 6 and describes how authorization policies can be dynamically managed & updated by multiple distributed dynamically allocated administrators. Section 8 describes how policies (especially privacy policies) can be "stuck" to information, and transported with the information throughout a distributed system. Section 9 briefly introduces the event management infrastructure which is used to support the passing of messages between system components, via the publish and subscribe paradigm, which is described more fully in D8.2. Section 10 describes the ontology for authorization and privacy policies. Section 11 concludes by describing the current limitations in the design to date, and indicating where further work will be done in future iterations of this deliverable, and where future research may still be needed at the end of the TAS^3 project. Section 11 also includes details of the standardization work that we have undertaken in the TAS^3 project in order to ensure that the authorization infrastructure is not only built on existing standards, but also contributes to future standards in this area.

1 Introduction

The overall objectives of WP7, as stated in the Technical Annex of the TAS³ project [1] are to:

- build a fully dynamic authorization infrastructure that allows credentials to be dynamically created and delegated between users and administrators, and policies to be dynamically managed and updated
- incorporate sophisticated real-life authorization requirements such as Break the Glass policies, dynamic separation of duties, state based decision making and adaptive audit controls
- contribute to international standards development in the area of IdM and authorization protocols and profiles and authorization ontology

The above overall objectives have further been enumerated into the following set of specific objectives:

- allow context dependent and user-controlled credentials to be dynamically created and user controlled credentials to be dynamically delegated between the users,
- allow context dependent credentials and user-controlled credentials to be retrieved, aggregated and validated in dynamically changing environments, even when the user is known by different identities at the different attribute authorities;
- make authorization decisions based on multiple policies, written in different policy languages & provided by multiple policy authorities including the data privacy subject;
- allow authorization policies to be dynamically managed & updated by multiple distributed dynamically allocated administrators;
- support Break the Glass policies which will allow the normal authorization policies to be over-ridden in emergency situations;
- integrate adaptive audit controls into the authorization infrastructure;
- allow history (or state) based authorization decision making to take place in Sun's XACML PDP;
- define an ontology for authorization policies and have it quality assured;
- contribute to open standards development within the scope of this work package;
- ensure that the outputs of this work package are seamlessly integrated into the outputs of all the other work packages.

These objectives will be achieved by using the latest state of the art technologies and standards and contributing to their further development. This document represents the first major revision of the official deliverable D7.1, whose purpose is to describe the design of the identity management, authentication and authorization infrastructure (hereby abbreviated to IdMAA) which is needed in order to achieve the above objectives. This version is now aligned with the architecture deliverable D2.1 [27], which was not published when the original version of this deliverable was issued. As the reader will observe, figure 2.3 from D2.1 shows the four callouts to

the authorization infrastructure when two web services communicate with each other, as does figure 2.2 in this document. The purpose of this document is to describe all the components that comprise this authorization infrastructure.

IMPORTANT CAVEAT. *Because this is the second iteration of the design of an extremely complex authorization infrastructure that has never been created before, this iteration may still contain bugs, flaws, omissions or extremely difficult to implement features that will have to be re-designed in the final iteration of this document. The reader should be aware that this document represents the output from only the first twenty four months of a four year project, and implementation experience gained in subsequent years of the project will further ensure that this iteration can be improved in the final version. Evaluation of the infrastructure is therefore not included in this design document.*

1.1 Research Contributions

The specific research contributions of WP7 to date are as follows:

- devised a new conceptual and functional component, the application independent policy enforcement point (AIPEP), whose purpose is to offload as much work as possible from the application dependent policy enforcement point (PEP) thereby making it easier for applications to integrate the TAS^3 authorization infrastructure;
- devised a new conceptual and functional component the master policy decision point (Master PDP), whose purpose is to call multiple subordinate PDPs which each support different policy languages, and to resolve any conflicts between their various authorization decisions;
- devised a new conceptual model and infrastructure for aggregating a user's attributes together, based on minor enhancements to existing standard protocols and a new conceptual and functional component, the Linking Service, whose purpose is to link together a user's different Identity Provider accounts;
- devised an infrastructure for the distributed enforcement of multiple obligations, which supports the enforcement of obligations written in any obligation policy language through the definition of a standard "obligations" interface;
- devised a mechanism for carrying sticky policies with application data and for incorporating these policies into existing standard authorization protocols;
- defined a new access control model called break-the-glass role based access controls (BTG-RBAC), and shown how this can be implemented using existing off the shelf stateless PDPs;

Future research contributions are expected to be in the area of delegation of authority, obligation enforcement and the dynamic management of policies.

1.2 Document Structure

This rest of this document is structured as follows. Section 2 describes the overall architecture of the IdMAA and includes the TAS³ obligations handling infrastructure. The obligations handling infrastructure is a core component that will occur several times in different components of the IdMAA. Section 3 describes the design of the Break the Glass (BTG) infrastructure in more detail, which will allow users who are not normally authorized to access resources, to gain access after "breaking the glass". This section also describes how adaptive audit controls can be supported in order to support BTG policies. Both of these rely on the obligations handling infrastructure described in Section 2. Section 4 describes the design of the credential aggregation infrastructure in more detail, where user credentials can be retrieved, aggregated and validated in dynamically changing environments, even when the user is known by different identities at the different attribute authorities. Section 5 describes the multiple policy authorization evaluation infrastructure in more detail, which will provide support for multiple authorization policies written in different languages to be evaluated and any conflicts between them to be resolved before the user is granted access to a resource. Section 6 describes the design of the infrastructure for the dynamic delegation of credentials between the various actors of the system. Section 7 builds on section 6 and describes how authorization policies can be dynamically managed & updated by multiple distributed dynamically allocated administrators. Section 8 describes how policies can be "stuck" to information, and transported with the information throughout the distributed system. Section 9 describes the event management infrastructure which is used to support the passing of messages between system components, via the publish and subscribe paradigm. Section 10 describes the ontology for authorization and privacy policies. Section 11 concludes this deliverable, by describing the current limitations in the design to date, where further work will be done in future iterations of this deliverable, and where future research will still be needed. It also includes details of the standardization work that we have undertaken in the project in order to ensure that the authorization infrastructure is not only built on existing standards, but also contributes to future standards in this area.

2 Overall Architectural Design of the Identity Management, Authentication and Authorization (IdMAA) Infrastructure

The design of the IdMAA infrastructure is a subset of the overall TAS^3 architectural design presented in Deliverable D2.1 TAS^3 Architecture [27]. As such each functional component of the IdMAA is designed to be location independent: it may reside in the same system as other functional components of the IdMAA or in a separate system of its own. There are security and performance implications related to how distributed the IdMAA infrastructure might be. Different communications protocols will be needed depending upon whether the components are closely coupled together in the same system, are distributed over a trusted network, or are distributed over the Internet, but this issue is not addressed here. It is addressed in Deliverable D2.4 Section 1.2 “Composition and Co-location of Architectural Components” [46] as this is a generic issue pertinent to the entire TAS^3 architecture.

2.1 Federated Identity Management

2.1.1 Identities, Identifiers and Attributes

A person’s identity is made up from a whole series of attributes that characterise him or her[1], such as: their physical characteristics and appearance, their past and present behaviours and reputation, their qualifications and group memberships, the names and identifiers used to label them etc. These identity attributes can be used by service providers (SPs) to grant or deny individuals access to their resources, for example, students from the University of Kent may download journals from the library. Unfortunately (or perhaps fortunately from a privacy perspective) no single person or system knows anyone’s complete set of identity attributes, although individuals are most likely to know most of the attributes that serve to identify them. Even then, there are limitations in this, for example, individuals might not know how much others trust them. Invariably then, computer systems typically only hold the partial identities of people i.e. a subset of their digital identity attributes. These computer systems are known as Attribute Authorities (AAs) or Identity Providers (IdPs)[2].

Before proceeding further, we should clarify the difference between an *identifier* and an *identity*. An identifier serves to uniquely identify an individual within one domain or system, as

[1] In order to be gender neutral, we will use “them” to refer to him or her in future. We will use *he* in the remainder of this document when we need to refer to a single person, but the person may be male or female.

[2] The difference between an AA and an IdP is that an IdP is an AA that also has the ability to authenticate the user so that the user can login and request his attribute assertions be issued to him in the form of digitally signed authorisation credentials

no two individuals within a system can have the same identifier. However, this identifier is only one of the identity attributes that comprise that person's digital identity within the system. Different computer systems know different subset's of a person's identity attributes, but each computer system will have its own identifier which uniquely identifies this individual within this system. An individual whose identity is distributed throughout many systems will therefore have multiple identifiers such as: their passport number, login ID, social security number, national ID, email address etc., which are each unique within their own systems. Some systems may store the identifiers from remote systems, as well as their own. For privacy (and other) reasons, users are typically wary about releasing their identifiers to third parties, since these can uniquely identify them, allowing SPs to merge identity information that the user perhaps wanted to keep private and separate. Other identity attributes, such as age, usually cannot be used to merge partial identities since they typically apply to many users (unless the anonymity set is too small). Identifiers are therefore rather special identity attributes since on their own they can always uniquely identify the user. It is therefore preferable to construct identity management systems where these permanent identifiers are not transferred between systems or domains. One of the less obvious privacy threats is that many innocent looking attributes may in fact become defacto identifiers. The email address attribute is one obvious example. Usually an attribute becomes an identifier when it allows a user to be individually picked from the anonymity set. i.e. if the anonymity set is sufficiently small then almost any attribute can be an identifier.

2.2 Authoritative Sources

Usually attributes have to be conferred on individuals by authoritative sources, known as Attribute Authorities (AAs). Whilst people may be trusted in some situations to assert some of their identity attributes themselves, for example, their favourite drink, they certainly won't be trusted in all situations to assert all of their identity attributes themselves, for example, their qualifications or criminal record. Thus different authoritative sources are usually responsible for assigning different attributes to individuals. For example, the university that one graduated from is the authoritative source of one's undergraduate degree attribute. An identity provider (IdP) is an attribute authority combined with a user authentication service, so that it can authenticate the user, and then issue the user with a digitally signed attribute assertion. When a relying party is presented with attributes without a clear (fixed) Authoritative Source it can use credential based Trust Management to obtain "trusted" attribute values.

Authoritative sources may remove attributes as well as assign them. For example, a university may remove a degree from a student, if it was subsequently proved that the student had committed plagiarism in their dissertation. Similarly, in the UK, the General Medical Council is the only authoritative source of who is a doctor, and it keeps a register of them. If a doctor commits malpractice, the doctor may be *struck off* the register by the GMC. Thus in federated identity management systems, we cannot rely on the individual to assert his various attributes, otherwise he might lie about his various roles and capabilities, and omit to tell about

negative attributes such as the points on his driving license. Similarly we cannot rely on a single identity provider to assert all a user's attributes, but only the ones they are authoritative for. For example, a credit card company would not normally be trusted to assert someone's degree qualification attribute. Consequently a set of authoritative sources may need to be consulted by service providers before the latter grant users access to their resources.

2.3 Authentication in FIM

In a centralised system, the user typically presents their identifier and an authentication token (such as a password or digital signature) to prove that they are entitled to be known by this identifier. The system then associates the user with this identifier and with all the attributes that it holds with this identifier. In a distributed system the user would typically have different identifiers in each local system, so if the user authenticated to one identity provider using his local identifier, this identifier would not be known by and therefore could not be used by the other local systems to grant the user access. When X.509 based PKI systems were first designed, they tried to solve this problem by allocating each user a globally unique identifier (called an X.500 distinguished name), which would be known by all local systems and therefore could be used to grant the user access. Since this global identifier was bound to the user's public key in an X.509 public key certificate, a signature created by the user's private key could be used as an authentication token by each local system. One of the reasons this X.509 based identity management system failed was the privacy concerns about everyone knowing everyone else's globally unique identifier.

The breakthrough came when it was realised that a user's identifier did not need to be globally unique, but could remain local to the system that allocated it. Authorisation to use a remote federated system could be granted based on the user's identity attributes attached to a pseudonymous identifier, rather than on the use of the user's permanent identifier (global or local). If the identity attributes are provided by a trusted authoritative source, after the user has successfully authenticated to it, then a service provider (SP) can be assured of the identity of the user, even if the user's pseudonym was previously unknown to it. The use of different pseudonyms with each SP is privacy preserving as no SP receives a correlation handle enabling it to combine the user's identity information that it collects with those that the other SPs collect. Hence systems such as Shibboleth [1], SAMLv2 [11] and CardSpace [2] were born. TAS^3 uses this model for authentication and authorisation.

However there is still one issue that is causing dissent within the worldwide community and this concerns the use of a pseudonym attribute which uniquely identifies the user to the SP. Suppose a SP wishes to know it is the same user that is contacting it every time so that it can provide a personalised service. If the user uses the same IdP every time, then the IdP needs to provide a set of attributes that uniquely identify this user from the set of all users that the IdP

authenticates. The easiest way to do this is for the IdP to create a new identifier attribute that uniquely identifies this user to this SP, i.e. a pseudonym, and to send this attribute to the SP each time the user authenticates with the IdP. The attribute value is unique to the IdP-SP relationship. The user will have a different pseudonym value for the same IdP with a different SP, so that the administrators of the SPs cannot collude and compare the attribute values in order to share information about the user, without the user's consent. The issue of dissent is this: should this identifier attribute be sent as an attribute assertion, along with the other attribute assertions (i.e. the identifier attribute is treated the same as all of the user's other attributes), or should the identifier attribute be sent separately in the authentication assertion as the attribute which uniquely identifies the user (i.e. the identifier attribute is treated differently to the user's other attributes)? CardSpace uses the former model as it does not have an authentication assertion. Liberty Alliance (built on SAMLv2) uses the latter model. Shibboleth (also built on SAMLv2) is somewhat ambiguous and uses both models in different parts of its documentation. Shibboleth may use a random identifier in the SAMLv2 authentication assertion and a pseudonym attribute in the attribute assertion, or it may use the latter in the authentication assertion. The SAMLv2 protocol can in fact support both models, and it is a profiling issue to determine which model an application should use with the SAMLv2 protocol. The TAS^3 project is also split on this issue. The attribute aggregation feature uses a random identifier in the authentication assertion whilst the single sign on feature uses a permanent pseudonym attribute in the authentication assertion.

Normally the user will authenticate to an IdP, and the IdP will issue the user with a digitally signed attribute assertion, which the SP can trust and use for authorisation. Note that TAS^3 is not concerned about the mechanism the IdP uses to authenticate the user (though it should be appropriately secure and unspoofable, following the best practises taking in consideration the risk and convenience required for adequate adoption). Consequently it is a local matter between the IdP and the user, and TAS^3 will not concern itself with authentication issues, other than to use the level of authentication assurance (LoA) (see section 4.1.2.) as a means of the IdP informing the SP about the level of authentication that was used. Alternatively, the user may authenticate directly to the SP, and the SP may then ask the IdP/AA for digitally signed attribute assertions about the user. In TAS^3 we use the former model since it relieves the SP from the burden of authentication, although we also support the second model when performing attribute aggregation.

2.4 Authorization in FIM

The authorization model paradigm that we have adopted is the well known "Subject – Action – Target" paradigm in which a subject attempts to perform some action on some target resource. We use an enhancement of the ISO Attribute Based Access Control (ABAC) model [2] (see below), in which the subject is granted or denied access to the target resource based on the attributes he

possesses. The reason for using ABAC is that it is more scalable and more manageable than a traditional discretionary access control system; for example, it is the model used for Internet shopping with a credit card. Each subject represents a real world principal, which is the action performer. Whilst a subject is often referred to as a user, subjects are not limited to human beings. Action is the operation that is requested to be performed on the target. It can be either a simple operation, or a bundle of complex operations that is provided as an integrated set. Target is the object of the action, over which the action is to be performed. A target represents one or more resources that need to be protected from unauthorized access. In TAS3 a target can be specified explicitly by way of <TargetIdentity> in the SOAP header. If it is not specified this way, then by convention the resource identified by the <Token> element of the <wsse:Security> SOAP header is the target i.e. the default target is the calling user's own resource.

In centralised ABAC systems, it is the same system that assigns attributes to users as assigns permissions to attributes and grants users access to resources, so there is implicit trust in the users' attributes. However, in federated identity management systems, the systems that assign attributes to users (i.e. the identity providers) are different from and remote to the systems that consume these and grant access to users (i.e. service providers). Thus trust needs to be established between identity providers and service providers.

Users are typically assigned attributes by various attribute authorities (AAs), for example, a degree attribute is assigned by a university, a driving license attribute is assigned by a government driving license authority, an employee attribute is assigned by an employer. Users can also be their own authorities for some of their attributes, for example, *my favourite drink* attribute. Users may claim various attributes when trying to access a target resource, but in a web services distributed system world we cannot assume that all the attributes claimed by a user are rightfully his. Consequently TAS3 has enhanced the ISO ABAC model so that subject attributes are presented as digitally signed attribute assertions (which we call authorization credentials) issued to the subject by one or more trusted (in the eyes of the resource owner) Attribute Authorities (AAs) or Identity Providers (IdPs). These trusted issuers are the authoritative sources of subject attributes. We call the service of issuing subject attributes, an authorization Credential Issuing Service (CIS). A corresponding authorization Credential Validation Service (CVS) is introduced at the target site to validate these credentials and determine which of the attributes can rightfully be claimed by the subject (see Section 6.4). Each resource owner (called the Target Source of Authority in Figure 2.1) specifies the credential validation policy which controls an aspect of gaining access to his resources.

ABAC is a generalization of the well known role based access control (RBAC) model [3], in which a role is not restricted to an organizational role, but can be any attribute of the subject, such as a professional qualification or his current level of authentication (LoA) [4]. Throughout this document when we will refer to a subject's roles, we mean any attributes that have been assigned to a subject. A subject can be the member of zero, one or multiple roles at the same time.

Conversely, a role can have zero, one or more subject occupants at the same time. In RBAC a role is associated with a set of privileges, where each privilege is the right to perform a particular action on a particular target. The TAS3 model is more flexible and allows sets of privileges to be assigned to sets of roles, rather than to single roles, since the latter is too restrictive in practice. For example if project managers have some organizational based privileges, project members have some project specific privileges, and project managers have some higher level project specific privileges, then without the ability to assign the latter to a combination of roles (project member and project manager), a new set of roles have to be specially created for each project manager. Thus each subject is authorised to perform the actions corresponding to his role memberships. Changing the privileges allocated to a set of roles will affect all subjects who are members of the role set (or who have been assigned the set of attributes).

The TAS3 model supports hierarchical A/RBAC in which roles (or attributes) may be organized in a partial hierarchy, with some being superior to others. A superior role inherits all the privileges allocated to its subordinate roles. For example, if the role Staff is subordinate to Manager, then the Manager role will inherit the privileges allocated to the Staff role. A member of the Manager role can perform operations explicitly authorized to Managers as well as operations authorised to Staff. The inheritance of privileges from subordinate roles is recursive, thus a role r_o will inherit privileges from all its direct subordinate roles r_s, and indirect subordinate roles which are direct or indirect subordinate roles of r_s. Role hierarchies need not apply only to organizational roles, but can apply to any attribute, such as level of authentication or assurance (LoA) as defined by NIST [4], where there is a natural precedence in the attribute values, in which a higher value implies the privileges of the lower values. In the LoA case, a user who has been authenticated to LoA value 4 (the highest) can be assumed to inherit the privileges assigned to the lower levels of authentication.

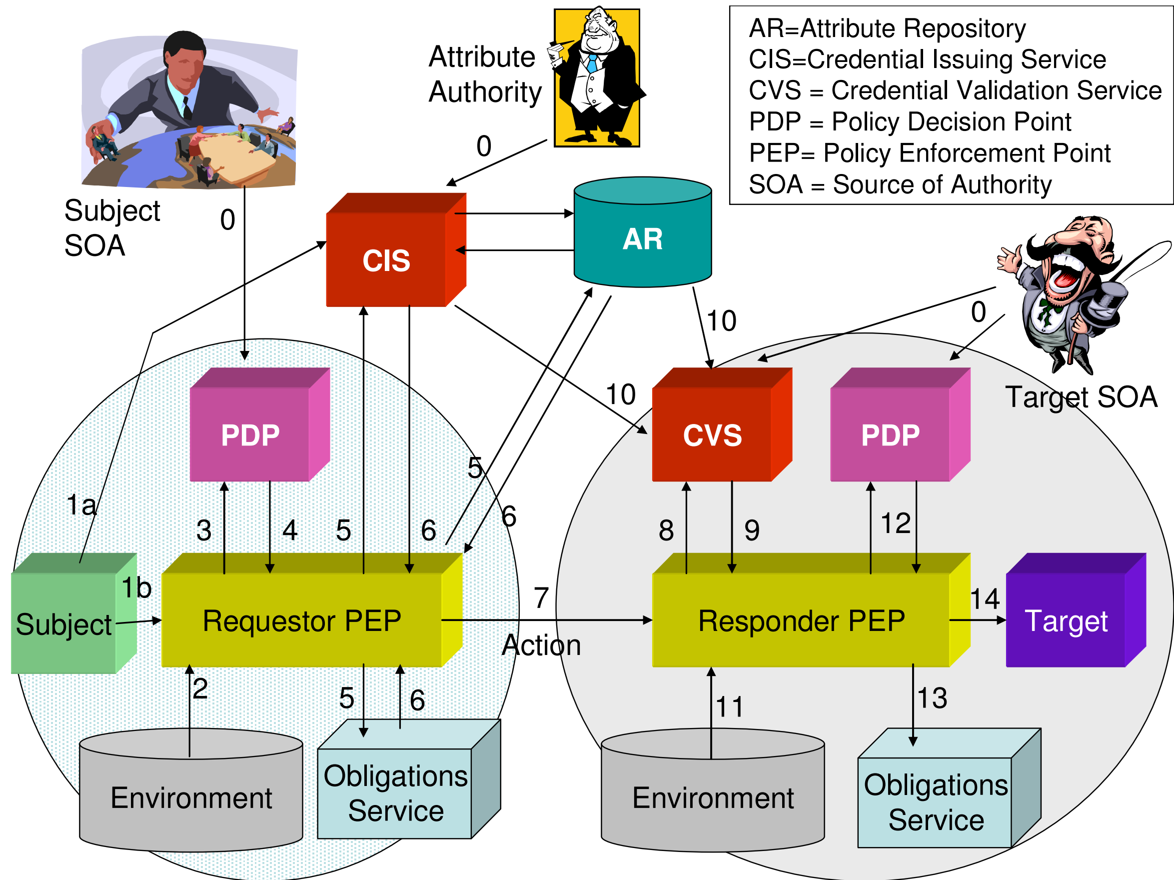


Figure 2.1. Request Authorization Model

Figure 2.1 shows our high level conceptual model for the TAS³ authorization infrastructure when a subject issues an action request to access a remote target. Step 0 is the initialization step for the infrastructure, when the policies are created and stored in the various components. Each subject may possess a set of credentials from many different Attribute Authorities (AAs), that may be pre-issued, long lived and stored in a repository (AR in Figure 2.1) or short lived and issued on demand (step 1a), according to their Credential Issuing Policies. Section 4 describes the design of the credential aggregation infrastructure in more detail, where user credentials can be retrieved, aggregated and validated in dynamically changing environments, even when the user is known by different identifiers at the different attribute authorities. The Subject Source of Authority (SOA) may dictate which of the subject's locally issued credentials can leave the subject domain for each target domain. When a subject issues an application request (step 1b), the requestor's application independent policy decision point (PDP) can inform the requestor's application's policy enforcement point (PEP) which credentials to include with the user's request (steps 3-4). These may be provided by the user along with his request (in step 1b after fetching

them in step 1a), or they may be collected from the Credential Issuing Service (CIS) or Attribute Repository by the requestor's PEP, either directly or with the help of an Obligations Service (steps 5-6). Obligations are actions that a PEP must enforce along with the decision returned by a PDP. The Obligations Service is the functional component that is responsible for enacting these obligations. It is described more fully in section 2.5 below. The user's request is transferred to the target site (step 7) where the target SOA has already initialized the Credential Validation Policy that says which credentials from which issuing AAs are trusted by the target site and to which local attributes they should be mapped, and the Access Control policy that says which privileges are given to which attributes. The user's credentials are first validated (step 8). This may require the CVS to pull additional credentials from an AA's repository or issuing service (step 10). The valid attributes are returned to the responder's PEP (step 9), combined with any environmental information, such as current date and time (step 11), and then passed to the responder's PDP for an access control decision (step 12). If the decision is granted the user's request is allowed by the responder's PEP (step 14), otherwise it is rejected. In either case, the responder's PDP may also return a set of obligations, which must be enforced along with the access control decision (step 13). In more sophisticated systems there may be a chain of PDPs that are called by a master PDP, with each PDP in the chain holding a different policy possibly written by a different SOA and possibly written in a different policy language (see section 5). In this case the master PDP needs to hold a policy combining policy written by the target SOA, which determines the ultimate response to give to the PEP based on the set of granted, denied or don't know responses returned by the chain of PDPs. Application PEPs however should be shielded from needing to know about this more sophisticated functionality.

Whilst Figure 2.1 shows many (though not all) of the components of the TAS³ authorization infrastructure, it does not show all the instances when applications may use it. It only depicts the case when a subject makes an outgoing action request to access a remote resource. In distributed workflow environments, when one service (acting on behalf of a user) makes use of an external service, we require the TAS³ authorization infrastructure to be called or involved four times by the application during the service invocation. The calling service should check if the outgoing call is authorised, the called service should check if the incoming service request is authorised and if the outgoing response is authorized, and finally the calling service should check if the incoming reply is authorised. This is shown in Figure 2.2. In this way we can ensure that all applications make full use of the TAS³ trusted infrastructure, and that all called and calling services can be sure that the other party is behaving in a trusted manner. Section 8 further describes this process, and how it can be utilised to enforce sticky policies.

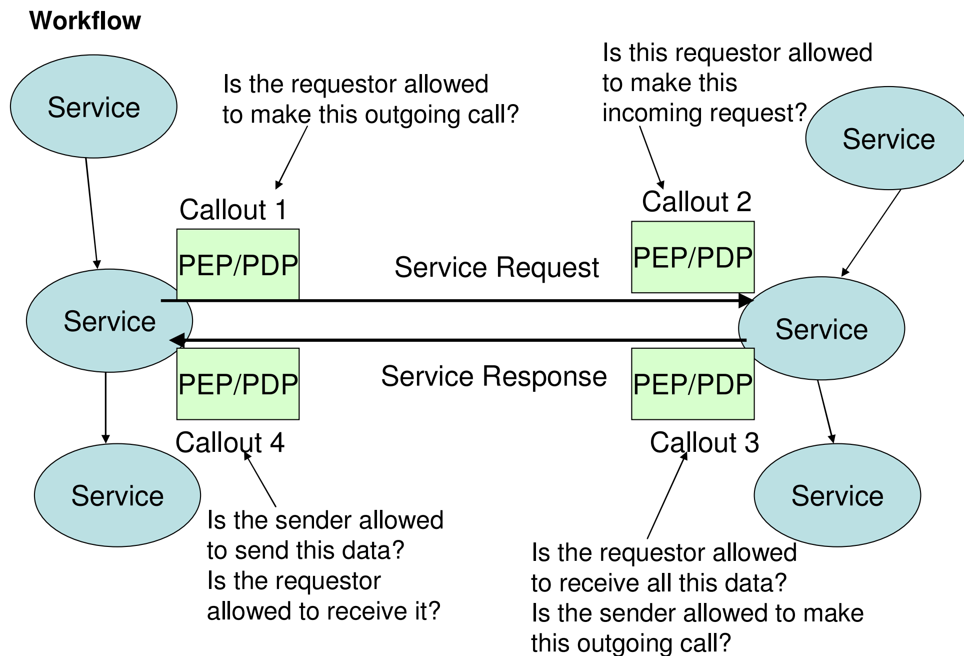


Figure 2.2 Application Callouts to the TAS³ Authorization Infrastructure

One objective that TAS³ would like to achieve is to make it as easy as possible for applications to implement the TAS³ trusted authorisation infrastructure. Therefore the more processing and code that can be removed from the application dependent PEP into the application independent infrastructure, the better. In current state of the art systems the only application independent code is the PDP which makes the authorisation decisions. For this reason, TAS³ has introduced the concept of an application independent PEP (AIPEP). The AIPEP will perform as many of the application independent security functions as possible, which cannot or need not be performed by the PDP. So the AIPEP may call the CVS instead of the PEP calling it, and the AIPEP will also process any obligations that can be enforced before the PEP is given the authorisation decision. This is shown in Figure 2.3.

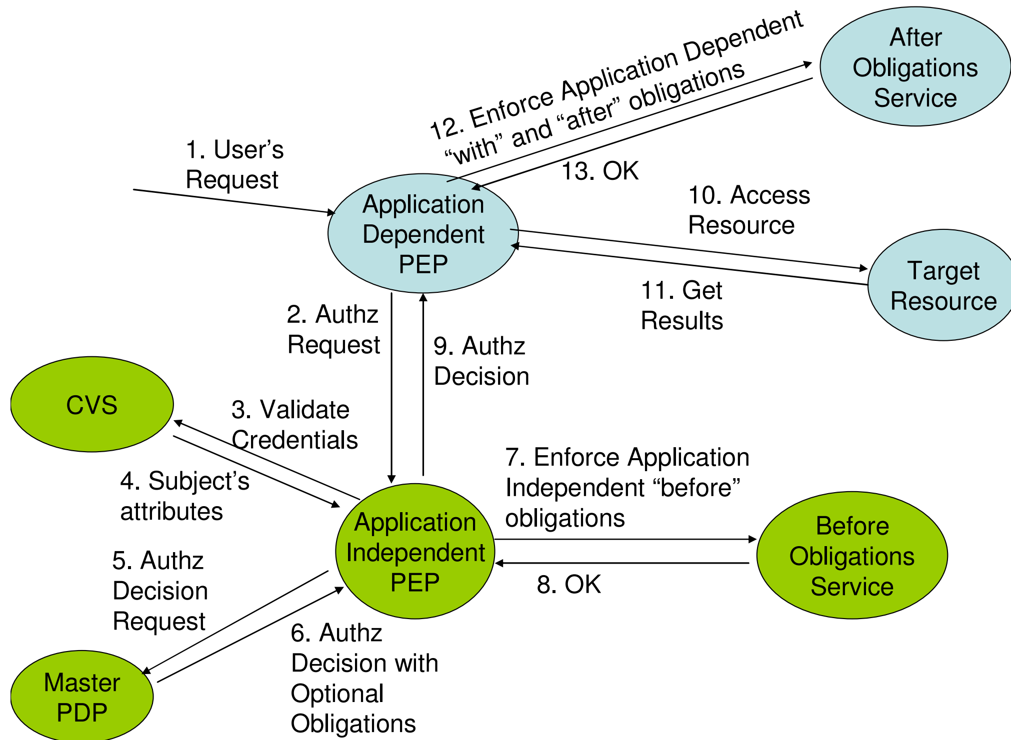


Figure 2.3. The AIPEP and Obligation Enforcement

2.5 The Obligation Enforcement Infrastructure

By introducing the architecture shown in Figure 2.3, we start to differentiate between the types of obligations that can be defined in the authorisation policies. This was first documented by Chadwick et al in [5], who described "before", "with" and "after" obligations. Some obligations need to be enforced before an access request is granted, e.g. before a user is given access the amount of logging needs to be increased to monitor what (s)he is doing. We define the *temporal type* of these obligations as *before* obligations. Some obligations need to be enforced after the user has been given access e.g. record the amount of cpu time that was consumed by the user. These are defined as *after* obligations. Some obligations need to be enforced simultaneously with the user's access, e.g. decrement the user's account balance simultaneously with the user's access to the system to withdraw money. We define the *temporal type* of these obligations as *with* obligations. Since a *before* obligation is enacted before the user is given access, then if the user's action fails the *before* obligation will have already been enforced. So success of a *before* obligation does not ensure success of the user's action. Similarly, since an *after* obligation is enforced after the user's action is performed, then the user's action may complete and yet the *after* obligation

may not be successful (although this would be an exceptional occurrence). Therefore the success of the user's action does not guarantee the success of an *after* obligation. Only a *with* obligation will succeed if the user's action succeeds and will fail if the user's action fails. If the user's action fails after partial completion, then the effect of any *with* obligations on the system needs to be rolled back to the state before the user was granted access. Only when the user's action succeeds can the *with* obligations be told to commit. So the *with* obligations and the user’s action need to be treated as an atomic action. Implementing atomic actions is a well known problem e.g. when updating databases, and one way of implementing atomic actions is to support two-phase-commit. A similar concept to the temporal type is now a feature of the latest standardisation work in XACMLv3 [6] and TAS3 will be contributing to this standardisation work.

Another way of classifying obligations is whether they are reversible or not. For example, an obligation that causes an email to be sent is not reversible, whereas one which renames a file potentially is reversible. How this might be used in the obligations infrastructure is currently for further study.

In order to enforce a diverse set of obligations, we introduce the Obligations Service which coordinates the enforcement of the set of obligations. Each obligation is enforced by an obligation specific Obligation Service, which is registered with the Obligations Service at program initialisation. Each Obligation Service has the same defined interface so that the Obligations Service does not need to be concerned about the specifics of any obligation. The conceptual interface is specified in section 2.5.1. Since the TAS3 authorisation infrastructure reference implementation is written in Java, we provide the Java interface for an obligation service in Appendix 1. Any application that provides its own Java object that conforms to this interface can have its obligation executed by the Obligations Service. The Obligations Service will be able to process this obligation along with all the other obligations, regardless of either the action the obligation will perform or the language used to specify the obligation (or obligation policy to be more precise). The language for specifying obligation policies now becomes a private matter for the user interface that defines it and the obligation service object that enforces it. The obligation policy itself can be carried around the TAS3 infrastructure along with the other sticky policies and eventually it will end up being passed to the correct obligation service object that understands the language it is written in. All that this requires is a minimum XML wrapper to be placed around each obligation policy specification, so that it can be transparently included into the XACML protocol which already has defined place holders for obligation policies. This XML wrapper is presented in Appendix 1.

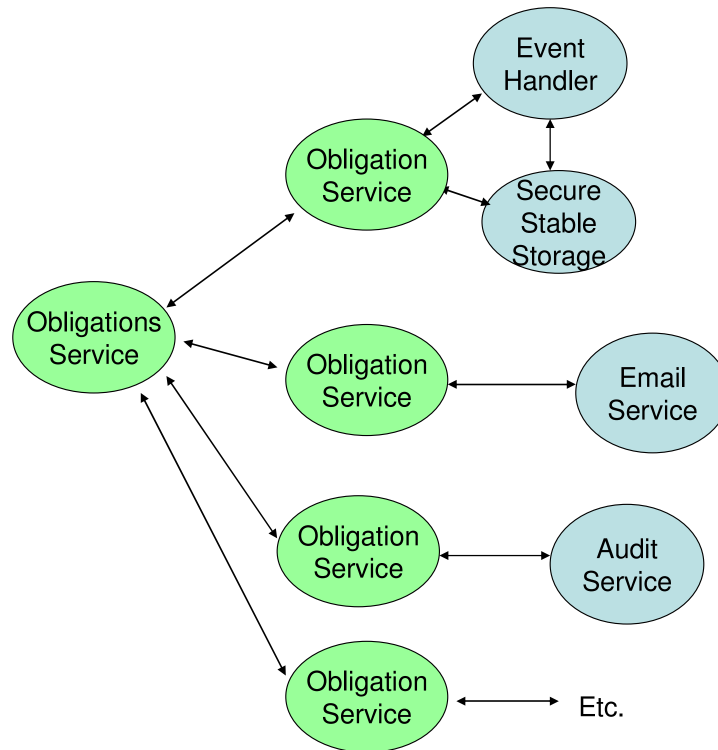


Figure 2.4. The Obligations Handling Service

In the TAS[3] project we will provide a number of general purpose Obligation Services that can be used by many different applications, such as: an Email obligation service that can be used to notify a person about an event, an auditControl obligation service that will adapt the amount of auditing information that is recorded (see section 3.3), and a delete file obligation service that can be used to ensure that a file has been deleted after a period of time (e.g. to enforce privacy policies). As Figure 2.3 shows, the Obligations Service can be called from multiple places of the authorisation infrastructure. We will use the same Obligations Service again within a state-based PDP to implement Break the Glass policies as described in Section 3.

In a distributed environment it will sometimes be the case that an obligation raised in one system has to be enforced in a remote system, an example being a user who places a privacy policy on her PII, which morphs into an obligation policy when the PII is retrieved by a remote system. Figure 2.5 shows that the Obligations Services in the distributed system will indirectly communicate with each other via the sticky policies that are attached to the application data and passed around the TAS[3] network, as described in section 8. The PEP interceptors at each site will ensure that the obligation policies are stuck to the outgoing data and are unstuck from incoming data and passed to the authorisation infrastructure on receipt. If the incoming obligations cannot

be enforced then the AIPEP will return deny to the PEP, which will then refuse to accept the incoming data. An example scenario of such remote obligation enforcement is given in Appendix 3. The TAS³ infrastructure is being constructed so as to be capable of enforcing these types of obligation.

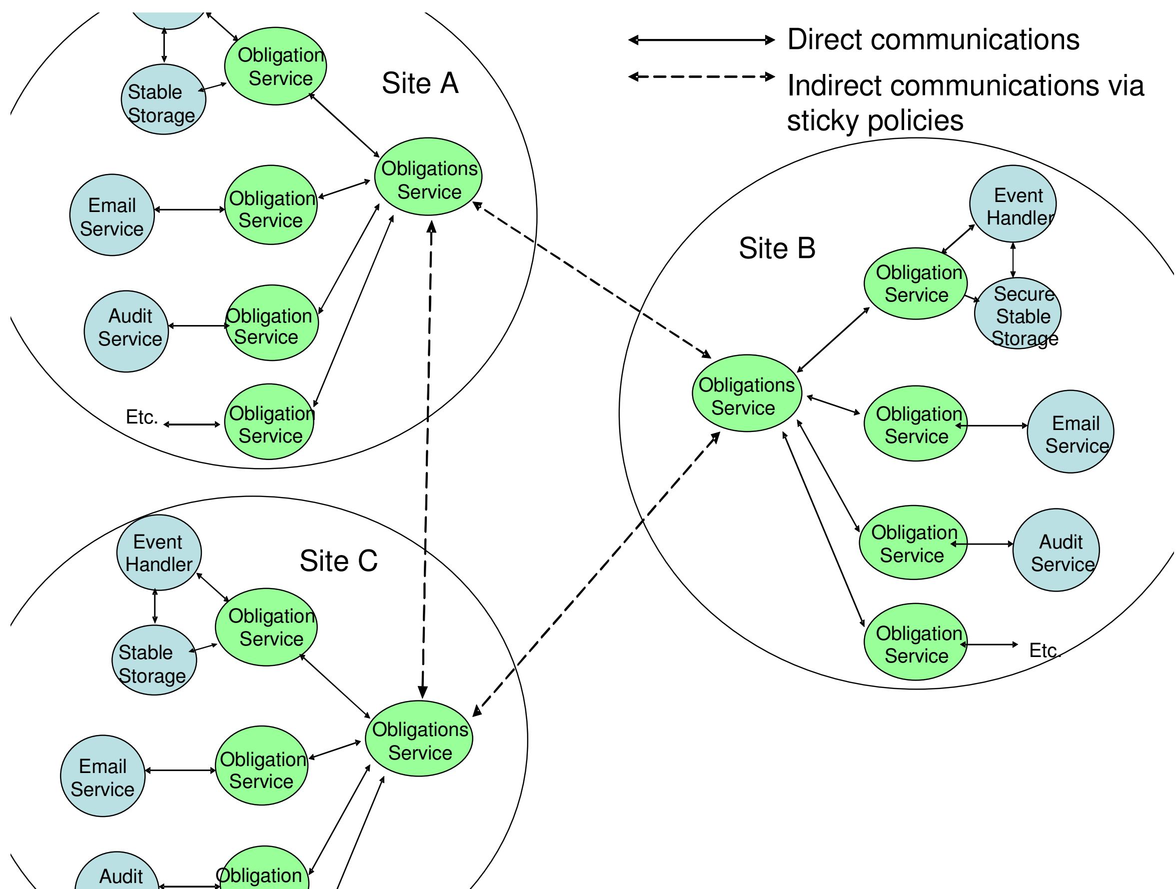


Figure 2.5. Distributed Handling of Obligations

In prior research [31] we have described how obligations can be integrated into the CORE RBAC model in a transparent and secure way. This augmented RBAC model is capable of providing obligations for both Grant and Deny responses and so it is a suitable model to use to integrate the BTG features as described in section 3.

2.5.1 The Obligation Service Interfaces

Each Obligation Service in the TAS³ framework has the following interfaces:

- The ObligationInformation Interface – this returns information about the specific obligation and its enclosed policy.
- The ObligationConstructor Interface – this is used to convert the content of the obligation (the obligation policy) into an enforceable Obligation object
- The Obligation Interface – this is used to execute the obligation

These interfaces are described in more detail below.

The **ObligationInformation** interface is used to retrieve information about a specific obligation.

- Each type of obligation must have a globally unique identifier which is used to uniquely identify the obligation type. A method named ***getObligationIdentifier*** is used to return the Uniform Resource Name (URN) of the obligation type, in the form of a String.
- Each obligation has to be enforced by an obligation subject. This is the component of the obligations infrastructure that invokes the Obligation interface. In our case it is one of the distributed Obligations Services that should enforce the obligation. A method named ***getObligationSubject*** is used to return the URI of the obligation subject (Obligations Service).
- Each obligation has a temporal type (i.e. before, with or after) as explained above. A method named ***getObligationTemporalType*** returns the temporal type as an enumerated type.
- Each obligations has content (or a policy specification) written in some language. The content of the obligation can be of any form in any language. We do not limit ourselves to any specific language or content, which helps to future proof our implementation. To get the content of the obligation we define a method named ***getObligationContent***. To make the design completely generic, the return type is itself an object, called the obligationContent. This object will subsequently be used to construct the executable object which will perform the obligation.
- Each obligation may have a fallback action or set of actions that should be taken in case the obligation fails to be enacted. These fallback actions will themselves be other obligations. This provides a way for recursion. We define a method named ***getFallBack*** with a return type of a list of ObligationInformation objects. This provides a way for an obligation to specify a set of fallback obligations that should be enacted if it fails to be enacted itself. This set could be a conjunctive or disjunctive list. The ideology behind this method is that if at the time of enforcement of the obligation it is found that it is not possible to enforce it, then this method will be called to find the alternate list of fallback obligations that should be enforced instead. If any of the fallback obligations fail then again this method will be called to get another alternate list of fallback obligations. This recursive process will continue until the fallback list is empty, either because all the fallback obligations have succeeded or because any fallback obligations which failed did not specify further fallback obligations to be taken. Clearly we have to be careful that

infinite recursion does not occur and a configurable control parameter will be built into the Obligations Service when it is constructed to limit the number of fallback obligations that will be executed.

The **ObligationConstructor** interface contains a method which is used to convert the content of an obligation into an enforceable Obligation object. For example the content of an obligation may provide instructions to send a particular e-mail message to a recipient, but it may not necessarily provide full details about the SMTP server to use, such as its DNS name/IP address and authentication credentials to use. The details of the SMTP server can be fixed and built into the construction of every obligation object of this type. This relieves the obligation policy writer from the burden of defining every single detail about the obligation enforcement when defining the obligation policy. Only variable parameters need to be part of the obligation policy. An obligation may require details about the current user's access request in order to correctly enforce the obligation, for example, the obligation may be designed to send an email to the current user's manager when the user Breaks the Glass. Therefore all the parameters from the authorisation decision request context will be passed to the obligation at construction time. The objectives of having the ObligationConstructor interface are 1. To validate that the obligation is actually enforceable before the enforcement of any obligation starts, and 2. To check (as far as possible) that all the necessary information and resources are available for enforcing the obligation so that the risk of failure is minimised.

- This interface has one method named ***construct***. It takes two parameters – an obligationInformation object and the request context object which was passed to the PDP for an authorisation decision (and which contains all the parameters of the authorisation decision request). The *construct* method returns the constructed Obligation object which can subsequently be executed as described below. This method throws the ObligationConstructionException when the Obligation can't be created, for example, when the ObligationConstructor doesn't recognise the obligation type, when there is missing information or there is a syntax error in the obligation content or there is a problem with resource allocation.

The **Obligation** interface is defined for executable obligation enforcement objects.

- It has a method ***doObligation*** for performing the actual obligation. For example the send e-mail Obligation will actually send the e-mail message when this method is called. Although checking is done when constructing the Obligation object so as to minimise the probability that doObligation will fail, nevertheless in a distributed environment we can't guarantee that when the doObligation is called it will always succeed. For example if a resource such as an SMTP server is available and checked at object construction time it could still be down when the doObligation method is called. An ObligationException is thrown by the doObligation method when the obligation fails to be enforced. This is the signal to the Obligations Service to take the fallback action.

- Suppose that a list of obligations need to be enforced. Suppose that some of the Obligation objects have already been constructed and have assigned resources to themselves. Suppose that construction of the next Obligation object fails. In this case we know that all the obligations cannot be enforced so we will not process of any of them. Consequently we will need to free the resources held by the already constructed Obligation objects. The method named ***freeResouces*** in defined in order to free any resources held by an Obligation object. After calling this method the Obligation object should become invalid and doObligation cannot be called on it.

2.5.2 The Obligations Service

The Obligations Service is responsible for sequentially calling all the obligation services, and ensuring that, as far as is possible, they are either all successfully completed or none are. As stated above, one of the reasons for constructing all the obligation objects before executing any of them, is to limit the possibilities of failure partway through. If all obligation objects are constructed successfully, then the execution of them starts. If after the first obligation has successfully completed, a subsequent one fails, this is when the obligations service attempts to execute the fallback obligations of the failed one rather than terminating part way through.

Each ObligationsService has a name, which is a URN. This URN is set during start up and configuration, and is used to match against the subject URN obtained from the ObligationInformation of each obligation. This allows the ObligationService to check whether it is responsible for enforcing this obligation or not. If the URNs do not match then the ObligationsService will bypass this obligation and move onto the next one returned by the PDP.

The ObligationsService object comprises two methods, one is addObligationConstructor which takes as parameters the obligation ID and ObligationConstructor of an obligation and binds them together so that the ObligationsService knows which obligation object to invoke for a particular obligation ID. The second method is *doObligations* which takes a to-be-enforced list of obligationInformation and the request context, and converts them into enforceable Obligation objects. It then enforces the obligations one by one.

The procedure for doObligations for *before* and *after* obligations is given below:

1. Check whether its own URN matches with the URN of the obligation in the to-be-enforced list, as obtained from obligationInformation. If the URNs match, then keep this obligation in the list, otherwise put it in the return list of un-enforced obligations.
2. Check whether it can recognise the unique obligation identifier of the obligation. If it can recognise the obligation ID continue otherwise return an exception “unrecognised obligation” along with the obligation ID.

3. Check the temporal type of the obligation with its configured set of temporal enforcement types. If it is in the set then move to the next obligation in the list and return to step 1. If they are different return an exception "wrong temporal type". When the complete to-be-enforced list has been processed without exception, continue.
4. Construct all the obligations in the list into enforceable Obligation objects. This is when resource allocation, syntax and all other input information checking for the obligation, is undertaken. If any of these constructions fail, then release the resources of the constructed obligations and return an exception "obligation <ID> failed to construct due to <reason>".
5. Enforce all the obligations one by one, by calling the doObligation method. If any obligation fails at this stage then call the getFallback method to get the list of alternate obligations to enforce. If no fallback obligations exist then move to the next obligation in the to-be-enforced list, otherwise construct the fallback obligations and execute these first. If any of these fail then call getFallback recursively until the fallback list is empty, then move to the next obligation in the to-be-enforced list.
6. Once the to-be-enforced and fallback lists are empty, return the list of un-enforced obligations to the caller.

The two-phase commit doObligations procedure for *with* obligations is still for further study, since this will need to be co-ordinated with the application's enforcement of the user's action. In the first implementation only *before* and *after* obligations will be enforced.

3 Break the Glass Infrastructure

Break the Glass (BTG) is a functionality that allows a user who is not authorised to access something, to do so in exceptional situations, on the full understanding that he will have to explain this to the appropriate authorities at a later point in time. BTG is required in at least the following scenarios: the successful mitigation of an emergency situation such as accessing a confidential patient record in a hospital accident and emergency department; the successful mitigation of an exceptional situation such as when the policy writer made a mistake and did not grant access to a user who should have been granted access and it takes a significant amount of time to alter the current active policy; and when the policy writer knows that some other users (but not precisely who) might wish to be granted access under emergency or exceptional situations and wants to be notified when this is the case so that he can monitor how often these situations occur.

We can model BTG policies by introducing a set of BTG state variables, BTGi, that are normally FALSE, but which switch to TRUE when a particular glass is broken by an authorised user. We can have a policy rule which says which users are authorised to set any particular BTGi state to TRUE for example emergency ward doctors. We can have another policy rule which says which users can access a resource when a particular BTGi state is TRUE, for example staff nurses, and a final rule which says which users can reset particular BTGi state variables to FALSE e.g. the site manager. The granularity of the BTGi state variables should be independent of the permissions within the system i.e. there could be one BTG variable covering all permissions in the system (least granular) or one BTGi variable per permission (most granular) or an intermediate number of BTGi variables that cover a range of permissions, as decided by the resource owner.

In order to integrate BTG within TAS[3] we introduce the BTG PDP. This is a stateful PDP which holds the state of each BTGi variable in the system. Initially each BTGi state is set to FALSE, but it can be set to TRUE if there is a policy rule that allows a user to perform the break the glass operation $O^{BTG(op)}$ on a particular resource for a particular operation op.

The BTG PDP is accessed via an enhanced *CheckAccess* procedure, which we have called *CheckBTGAccess*. It returns one of three decision values to the application: Grant, Deny or P^{BTG}. A P^{BTG} response indicates to the application that the user has permission to break the glass for the requested operation on the requested object. $O^{BTG(op)}$ is defined as the "break the glass" operation for the operation op on the resource object.

The possible results for *CheckBTGAccess* are:

(GRANT, 2^{OBLGS}) **IF** *there is a rule granting the user either the requested permission, or permission if the BTGi state is TRUE, and the BTGi state is actually TRUE*

(P^{BTG}) **IF** *there is a rule granting the user permission to break the glass for the requested permission*

(DENY, 2^{OBLGS}) ***Otherwise***

The consequence of the above is that the XACML response context will need enhancing to allow the P^{BTG} response to be carried from the BTG PDP to the caller (i.e. the Master PDP, AIPEP or PEP), since currently it only defines grant, deny, not applicable or indeterminate (i.e. error) responses. A short term fix/work around could be to encode the P^{BTG} response as a deny with a status code of P^{BTG}.

From the above results we can see that we have three classes of resource users in our BTG policies:

- class A) users who are authorised to access the resource regardless of the BTGi state,
- class B) users who are only authorised to access the resource if the BTGi state is TRUE,
- class C) users who are not authorised to access the resource regardless of the BTGi state.

We also have two classes of BTG users in our BTG policies:

- class T) users who are authorised to set the BTGi state to TRUE, and
- class F) users who are authorised to set the BTGi state to FALSE.

Typically class B) and class T) users will be the same group of people, but they need not always be, e.g. a nurse might be class B) and a ward sister might be class T), thereby providing some supervisory level of control over when a nurse can see confidential material. By keeping classes B) and T) separate, we allow this amount of flexibility. Class F) users will typically be managers and people in authority, who might reset the BTG state and then check that the BTG access was legitimate.

A typical example of the use of the BTG infrastructure is given in Figure 3.1.

- (Step 1) The user attempts to access a confidential resource which she is only allowed to access if the glass is broken.

- (Step 2 and 3) The user is authenticated.

- (Step 4) The application asks the BTG PDP if the user is allowed to access the resource. In the case where there is a policy rule granting access to the object, *CheckBTGAccess* returns Grant, so it goes to step 9. In the case where there is a policy rule granting O^{BTG} access to the object the BTG-RBAC engine returns P^{BTG} as the decision value. In all other cases CheckBTGAccess returns Deny and the request terminates here.

- (Step 5) Assuming P^{BTG} is returned in step 4, the application can now ask the user if he/she wants to O^{BTG} on that resource. If the user chooses to O^{BTG} (giving a reason for it, if applicable) goto step 6. In the case where the user chooses not to O^{BTG} the original request terminates here.

- (Step 6) The application calls the BTG-RBAC policy engine passing the session details, the requested operation ($O^{BTG(op)}$) and the requested object. The BTG-RBAC policy engine checks the policy, sees the operation is granted, sets the BTG_i state variable to TRUE and returns any obligations associated with the $O^{BTG(op)}$ operation (e.g. notify a responsible manager, write to an audit) to the application along with the Grant response.

- (Step 7) The application performs the returned obligations and the user is again shown the option to access the resource he requested and selects it.

- (Step 8) The application calls the BTG-RBAC policy engine again, passing the session details, the original requested operation and object. *CheckBTGAccess* returns Grant as the BTGi state variable is now set to TRUE.

- (Step 9) The application makes the requested operation to the resource

- (Step 10 and 11). The application receives the result and passes it to the user.

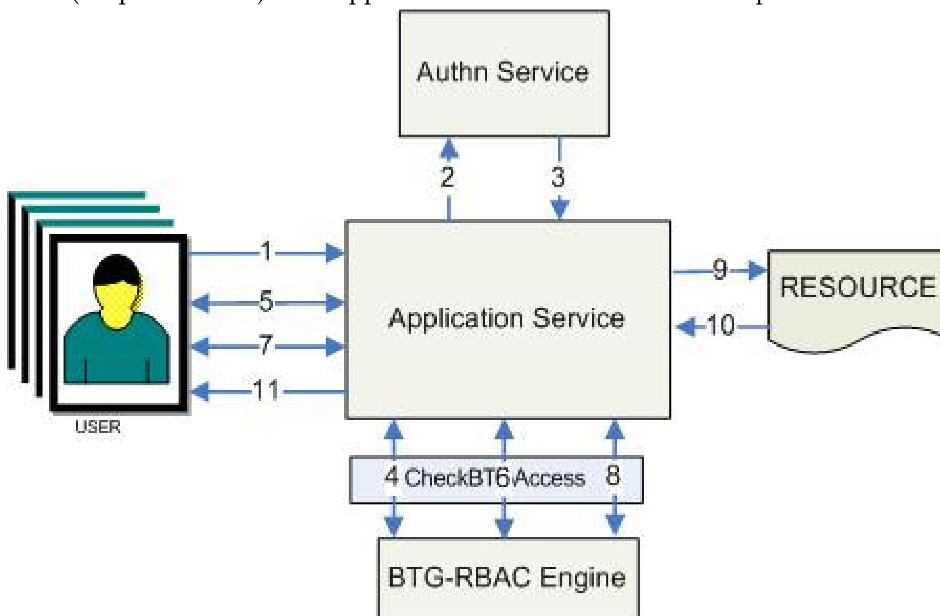


Figure 3.1. Using the Break the Glass PDP

An example of a BTG-RBAC policy is given in Table 3.1. BTGi is a state variable of n dimensions over subject, operation, object and environmental attributes taken from the request context i.e. BTG(s,op,ob,env) and will be described more fully in section 3.1 below. Every Table 3.1 states that Role *r1* is allowed to *read obs1*, Role *r2* is allowed to *read obs1* if the break the glass variable *BTGi is TRUE*, Role *r3* is allowed to *read obs1* if the break the glass variable *BTGi is TRUE* but the system must perform one obligation simultaneously with granting access, Role *r2* is allowed

to "break the glass" for *reading obs1* but the system must perform three obligations if *r2* does this, and Role *r4* is allowed to set the *BTGi state variable to FALSE*.

TABLE 3.1 – EXAMPLE OF A BTG-RBAC POLICY.

Role	Operation	Object	BTG variable	Obligations
r1	read	obs1		
r2	read	obs1	BTGi	
r2	$O^{BTG(read)}$	obs1		[Notify Manager; Write to Audit; Reset BTGi to FALSE after 30 mins]
r3	read	obs1	BTGi	[Write to Audit]
r4	$reset^{BTG}$	BTGi		

3.1 The BTGi state variable

Concurrently with a successful O^{BTG} operation there is the need to set the appropriate BTGi state variable to TRUE (if is not already set). The BTG model is consequently state based as it needs to remember the state of the various BTGi state variables. The writer of the BTG policy determines the dimensions of the BTGi state variables. They are based on the user's attributes, the operation, the object, and the environmental parameters such as a time period, etc. The actual values are taken from the access control decision request context. An example of various BTGi state variables and their dimensions is given in Table 3.2.

TABLE 3.2 – EXAMPLES OF BTGI STATE VARIABLES.

BTGi Policy	User Attribute	Operation	Object	Environment
BTG(role,op,ob,periodic30mins)	r2	read	obs1	9.00-9.30
BTG(,,ob,Daily)	*	*	obs1	3 Nov 2009
BTG(,op,ob,)	*	write	obs1	*
BTG(name,,,)	Fred	*	*	*

The BTG state variable in the first row of Table 3.2 is dependent upon all 4 dimensions. i.e. the role of the user, the operation, the object being accessed and a time period of 30 minutes. Because it is time dependent, the BTG PDP will automatically create a new (or replace an existing) state variable every 30 minutes. Whether this period is fixed to start either on the hour

every hour or when the user firsts breaks the glass is an implementation issue (we have implemented the latter). If a user with role *r2* requests to perform the *read* operation on object *obs1* at 9.12 am, and has BTG access as specified in the second row of Table 3.1, then if the current value of this BTGi variable (r2, read, obs1, 9.00-9.30) is TRUE he will be granted access, otherwise he will be denied access. However, because this user also has permission to break the glass (from row three in Table 3.1) then the BTG PDP will return P^{BTG} instead of deny.

The BTG state variable from the second row is only dependent on the object being accessed and the date i.e. it is the same variable for all operations by all users on a specific object on a specific date. The BTG PDP will automatically replace these state variables each day. If any user attempts to perform any operation on obs1 on 3 Nov 2009, the value of the variable BTG(obs1, 3 Nov 2009) will be consulted. If a user with role r2 or r3 attempts to read obs1 on this day, they will be granted permission if this variable is TRUE (from rows 2 and 4 of Table 3.1). If it is FALSE users with role r3 will be denied access but users with role r2 will be returned P^{BTG} instead of deny (from row 3 of table 3.1). Once this glass is broken by role r2, the state BTG(obs1, 3 Nov 2009) will be set to TRUE so that any user with any permission on *obs1* on this day if the glass is broken, will have had the glass broken for them.

The BTG state variable from the third row is only dependent on the object being accessed and the mode of access. If a user breaks the glass for writing to obs1, this will not affect any user with permission to read obs1 if the glass is broken.

The BTG state variable from the fourth row is only dependent upon the name of the user i.e. each user has their own BTG variable. If they break the glass for any operation on any object at any time, the glass remains broken for all their accesses to all objects until the BTG state has been reset to FALSE (see next section).

With the use of an n dimensional BTG state array, BTG can be defined in as fine-grained way as required so that the system can record BTG state with a combination of user roles and attributes, operations, objects and environmental parameters.

3.1.1 Resetting the BTGi State Variables

BTG state variables require a service that can reset each BTGi state variable to FALSE. This can be done automatically, semi-automatically or manually. All three ways are potentially needed. Automatic resetting means that the BTG PDP itself resets the BTGi state variable to FALSE after a specified event has occurred. The event must be specified by the administrator when creating the BTG policy. Example events could be the expiration of a time period such as 30 minutes (as in Table 3.2), or after a certain number of accesses have been made while the BTGi state was TRUE. Automatically resetting the BTGi state to FALSE controls the

availability of a resource once the glass has been broken, and requires a second breaking of the glass after the specified event has occurred, before additional accesses can be granted. We do not dictate how these events should be specified for the BTG PDP, or which events should be supported by a BTG PDP. We leave this to each BTG PDP supplier to specify for themselves. In our implementation we do not provide an automatic periodic resetting of BTG state variables, but rather we provide a semi-automatic resetting mechanism.

The semi-automatic resetting of the BTGi state is similar to automatic resetting, but it is carried out in a standardised way by a system component that is external to the BTG PDP. For this we specify a new function *resetBTGstate (BTGi)* that must be supported by the BTG PDPs. Any trusted system component may call this function to reset the *BTGi* variable to FALSE. In our implementation we use an obligation service as the trusted system component. Using obligations, the security administrator sets an obligation in the policy rule that describes when the BTG state is to be reset. The events for when this occurs can be similar to the ones for automatic resetting. For example, obligations could be defined as follows: *Obligation set BTGi to FALSE after 30 minutes* or *Obligation set BTGi to FALSE after 3 BTG accesses*. This obligation is returned within 2^{OBLGS} once the user chooses and is granted the right to perform O^{BTG}. The obligation will be performed when the event that is defined occurs (“after 30 minutes” or “after 3 BTG accesses”). The middle row of the example policy in Table 3.1 gives an example of an obligation that will reset the BTGi state to FALSE 30 minutes after it is set to TRUE.

Manually resetting the state means that human intervention must occur before the BTGi state is set to FALSE, and policy rules should specify who is allowed to reset the state. This requires a new operation for resetting the state, which we have defined as the $reset^{BTG}$ operation. This operates on the BTGi state variable as the resource object. The last row of the example policy in Table 3.1 gives an example of a policy rule for manually resetting the BTGi state to FALSE. The BTGi state will only be reset after the permitted role, *r4*, issues the $reset^{BTG}$ operation on the BTGi object.

3.2 Implementing BTG

Rather than taking an existing stateless PDP such as Sun’s XACML PDP, and modifying it to be stateful, instead we have chosen to implement the BTG state functionality in an interface layer that can placed above any stateless PDP. This is shown in Figure 3.2 below. The advantage of this approach is that any stateless PDP can be turned into a BTG stateful PDP by placing our BTG state layer between it and the caller. The interface that we use between the layers is the standard XACML request-response context, enhanced to support the P^{BTG} response. Thus the PEP, AIPEP or Master PDP can all call the BTG Stateful PDP or the stateless PDP directly, since they all support the same interface. The BTG state layer uses the Obligations Service described above to maintain the BTG state. There is a “BTG state” obligation service which

maintains the BTG state and thereby insulates the BTG state layer from the specific implementation details of recording and storing the BTG states. In our initial implementation, this obligation service uses memory only storage, although it could be replaced by a different obligation service that uses an external database if the long term stable storage of BTGi state is required. However we don't think this is essential in the reference implementation, since the memory only store is easier to implement and faster than a database to access and it is accessible for as long as the system is running. We do not think it is a great inconvenience to users if, after a crash or restart, they have to break the glass again because the BTGi state has been reset. Furthermore, if the storage requirements were to grow so large as to exceed the memory capacity of modern systems this most probably means that the original policy has flaws in it; having to break the glass is meant to be an infrequent event. Our implementation only keeps BTGi variables in memory that are TRUE, so once a variable has been reset to FALSE it doesn't take up any memory.

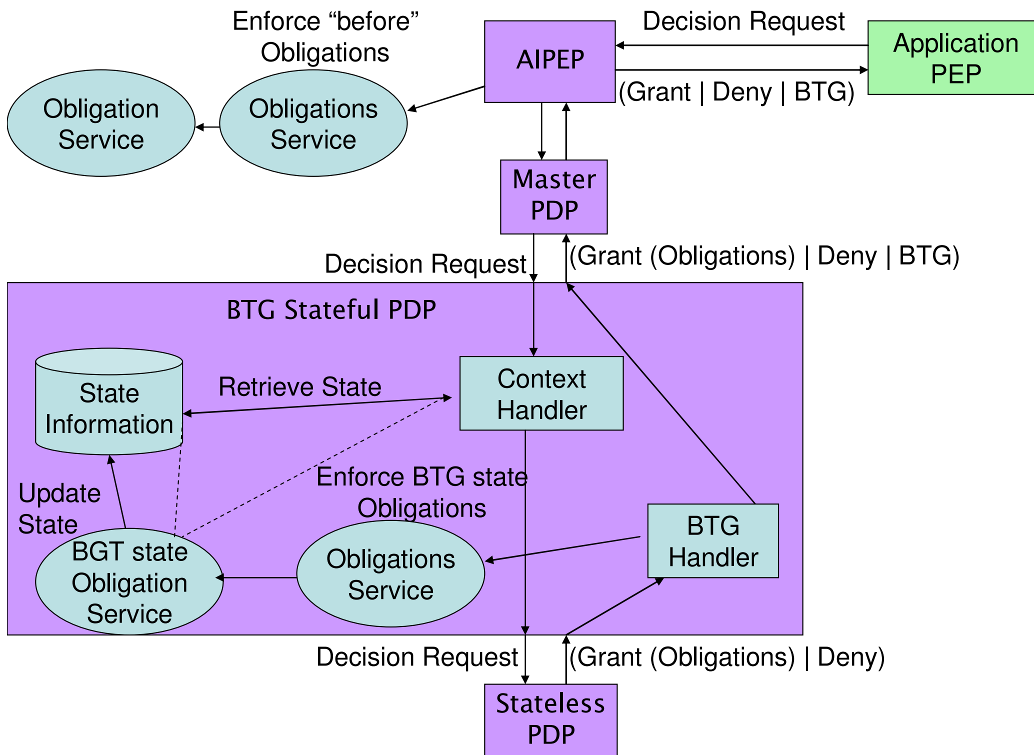


Figure 3.2. The BTG PDP Implementation

When the caller passes the request context to the BTG PDP, the first thing its Context Handler does is retrieve the relevant BTGi state variable from the state information store. In reality this is done by calling a retrieve method on the BTG state obligation service, rather than accessing the state directly as shown in Figure 3.2. This serves to insulate the BTG layer from the

actual implementation details of the state information. The relevant BTGi state variable is determined by matching the parameters of the request context with the BTGi state variables in the store and only returning those that match. The Context Handler adds the BTGi state(s)[3] to the request context and passes this to the stateless PDP for an access control decision. The policy rules that are configured into the stateless PDP are the same as in Table 3.1 except that the BTGi state is turned into a condition statement of the type IF BTGi is TRUE Then…..<existing PDP rule>. If the Stateless PDP returns any reply other than Grant, then this is passed back to the caller (Master PDP in Figure 3.2) unaltered. If the Stateless PDP returns Deny, then the BTG Handler creates a second request to the Stateless PDP, of the form "Can <the same User> perform operation $O^{BTG(\text{<same operation>})}$ on <the same Object>". The Stateless PDP has already been configured with the BTG policy rules as per row 3 of table 3.1, and it can therefore answer this question. If Deny is returned then the BTG PDP returns Deny to the caller. If Grant is returned, it will be accompanied by an optional set of obligations. The BTG Handler ignores these obligations, and instead returns P^{BTG} to the caller.

If the user decides that she wants to break the glass, then the PEP will issue the request "Can <the same User> perform operation $O^{BTG(\text{<same operation>})}$ on <the same Object>". The BTG PDP passes this request unaltered to the stateless PDP. The BTG PDP does not need to pick up any BTG state information for requests of this type, since we don't recognise the concept of being able to break the glass on a request to break the glass. If the user is Granted access to perform the BTG operation, then the response will be accompanied by an optional set of obligations. In this case, the BTG Handler now calls the Obligations Service to perform as many of the obligations as it can. Typically the BTG PDP Obligations Service will only be configured to known about the BTG state handling obligation, and other obligations, such as emailing the manager of the user, will be carried out by other Obligations Services such as those of the AIPEP or PEP. Our design of obligation handling, as described earlier, ensures that an authorisation system can have multiple Obligations Services, and each Obligations Service need only process a subset of the presented list of obligations. Thus the system administrator can decide which Obligations Service should best handle which obligation services.

Once the BTGi state has been updated to TRUE, the BTG PDP Obligations Service returns the set of unfulfilled obligations, and the BTG Handler places these in the response context to be passed to the caller. Eventually the response context is returned to the AIPEP, and this calls its Obligations Service to process the set of obligations that are in the response context. It is here

[3] It is possible that several BTGi state variables could match the request context, but in most cases we expect there to be zero or one match. For example, if the user has two roles, r1 and r2, and both are presented in the request context, and there are BTGi state variables for each role.

that typically the obligations such as emailing the manager of the user are enforced (though they need not be, they can be passed back to the PEP if desired. This is purely a configuration issue). Assuming that the AIPEP's Obligations Service processes all the BTG obligations, then the PEP is simply given a Granted response without any obligations. The PEP can now ask the user if she wishes to access the previously forbidden resource. Assuming the user wishes to do so, the PEP sends an access control decision request to the AIPEP asking if the user can access the resource. This is passed down to the BTG PDP, which fetches the relevant BTGi state from its store, and supplements the request context with this information. The Stateless PDP is now able to grant the user access to the resource, because the BTGi state variable is set to TRUE, meaning that the BTG condition on the access rule is now satisfied. The granted response is passed back up the chain to the PEP, and the user gains access to BTG protected resource.

3.2.1 Encoding the Permission to BTG response

The standard XACML request-response context does not have a "permission to break the glass" type of response. Consequently it needs to be enhanced to support the P^{BTG} response.

1. The ideal solution would be to standardise the BTG response as a fifth enumerated value for the <Decision> element, called BTG. This would make BTG an equally valid response as the existing Permit, Deny, Indeterminate and NotApplicable responses.
2. Alternatively, a less favourable solution would be to create a new Major status code called BTG. But this is something of a hack, both syntactically and semantically, since status codes are optional and the response is neither semantically a Permit or Deny but is genuinely intermediate to these. BTG is also neither Indeterminate nor NotApplicable, so which decision value would correctly accompany this Major status code?
3. The least favourable solution would be for the TAS3 project to invent its own minor status code and put BTG there without perturbing the XACML standard, but this would not provide BTG with any recognition outside of the project itself.

We have raised this topic with the OASIS XACML TC and the discussions to date have favoured a new standardised "BTG" XACML profile be written. It is very likely that during 2010 this will be done and we will be able to report the outcome of this in the next version of this deliverable.

3.3 Adaptive Audit Controls

The requirement, from a BTG perspective, is to increase the level of auditing from its current level to an application dependent higher level, from the time the user decides to BTG until the BTG variable has been reset to FALSE by the system or an administrator. Auditing should then resume at its original level. We can implement the increase (or decrease) in the level of auditing by having an *auditControl* obligation that is returned by the PDP at the same time as the

obligation to set the BTG variable to TRUE (or FALSE). This obligation will cause the application to increase (or decrease) its level of logging. Of course, once we have created an auditControl obligation service, it is not restricted to being used by only the BTG operations. It can be returned in response to a request by any user to access any resource. Thus we have a general purpose adaptive audit control obligation service.

The granularity of auditing is an application dependent issue, as is the way that audits are recorded. Whether the application will audit every action by every user at the same level of granularity, or individual actions by individual users at different levels of granularity, is decided by the application. What the application independent obligation layer will provide is a general purpose adaptive audit control obligation.

3.3.1 Implementing Adaptive Adaptive Audit Controls

In our exemplar implementation, we will provide a Java auditControl obligation service for Java applications that use the Log4J facility [44]. It will work as follows.

Inside a Java Virtual Machine (JVM) there can be any number of Log4J 'Loggers' available to the application. Each Logger has a name. The application addresses each Logger by using its name. The names of different loggers are hierarchical strings, so they can be anything. An application creates or retrieves a Logger using the getLogger method. So an application can access loggers by calling 'Logger.getLogger("Personal.Stijn")' or 'Logger.getLogger("Personal.David")' or 'Logger.getLogger("system")'. Each Logger has a level, so for instance the current level for the Logger "Personal.Stijn" could be DEBUG, while the current level for the Logger "Personal.David" could be INFO. Levels form a set hierarchy which is: TRACE < DEBUG < INFO < WARN < ERROR < FATAL where the lower the level, the more logging output is produced. Since all levels are comparable, it's always clear which one is the greater of two. When a Java application sends a message to a Logger, it specifies the level it wants it to be logged at. Only when the level of the message is greater or equal than the current level of the Logger does the message get logged. Finally, each Logger has a number of 'appenders' attached to it. Appenders determine where the log message gets sent. Examples of appenders are: 'standard output', 'a file', 'a socket' or in due course, the secure audit web service 'SAWS'.

We will provide a general purpose (Log4J inspired) auditControl obligation service that will allow an obligation policy to:

- identify the name of the logger to use (eg based on the user's name),
- specify the logging level to use either absolutely, (e.g. LogLevel=INFO), or relatively (eg. IncreaseLogging which will decrement the log level of the specified logger so that more output is produced),

- specify appenders that need to be added/removed. In particular this could be useful if we want add/remove secure auditing. Eg. AddAppender=SawsAppender.

This will provide a very flexible adaptive auditing function for any Java program, providing the application also uses the Log4J loggers. The obvious restriction is that this auditControl obligation implementation is limited to Java applications, but it does show other developers how adaptive auditing can be facilitated with obligations, and it does show other programmers how they might be able to adapt our implementation to use similar features that may be present in their chosen programming languages.

4 Attribute Aggregation Infrastructure

One of the current limitations of Shibboleth, CardSpace, Higgins and similar systems is that the user can only select one identity provider, and consequently only a limited subset of his identity attributes. For many web based services this is not enough. Consider wanting to buy a book from Amazon, and in order to get a discount you need to provide proof of your IEEE membership, asserted by IEEE, as well as proof you have a credit card, asserted by Visa. It is currently not possible to do this with CardSpace or Shibboleth since the user can only select one identity card or one identity provider. We need a mechanism to allow a user to select (or aggregate) attributes from multiple identity providers in a single service session, without necessarily having the burden of authenticating to each identity provider during the session. It is for these reasons that we introduce the concept of a Linking Service.

4.1 Conceptual Model

The TAS3 conceptual model for attribute aggregation assumes that the user is the best (and probably only) person to know who are the authoritative sources for all of his identity attributes. This is a reasonable assumption to make, since most people know who issue their plastic cards, passports, health cards, driving licenses, group memberships etc. We also know that privacy protection is important from a requirements survey that we carried our prior to the design of the TAS3 system [9]. We have therefore devised a new web service, called a *linking service*, whose purpose is to hold links between the user's various identity providers, and to do this without compromising the privacy of the user. Thus none of the user's (partial) identity providers know about any of the user's other ones. Furthermore, in order to fully protect the user's privacy, the linking service does not know who the user is, or what identity attributes they have. It only knows that some user has links with several different identity providers, and it holds these links on behalf of the user. When the user contacts a service provider for service provision, and they are redirected to their identity provider for authentication, the identity provider returns a pointer to the linking service in its response, which allows the service provider to aggregate the attributes from the various linked identity providers. The identity and service providers trust the linking service to hold these links securely, and to not divulge them to anyone except under the instructions of the user. The user is allowed to say which linked identity providers can be released to which service providers, through an identity provider link release policy (see below).

4.1.1 Link Registration Phase

Here's how the link registration works. The user goes to the web page of his preferred linking service (there can be any number of these on the web). The linking service displays a list of all the

identity providers with which it has already established trust relationships. The user selects one that they want to link to another one. The linking service redirects the user to the chosen identity provider, whereupon the user is asked to login and authenticate. The user authenticates using the identity provider's chosen method, by providing their identifier and authentication token. Upon successful authentication, the identity provider creates a random (but permanent) identifier for the user which is to be used solely with this linking service. The identity provider returns an authentication assertion to the linking service, containing this permanent ID. This assertion effectively says "I have authenticated this user, and they are to be known by you as Permanent ID x (PIDx)." When the linking service receives this message it creates a new link entry for the user in its linking table, assigning the user its own local identifier, say Fred, then displays the list of identity providers again. The user selects another one, is redirected there, authenticates, and the second identity provider returns a different permanent identifier, say PIDy, to the linking service. The linking service adds this link entry to its linking table. The user can perform this identity provider linking as many times as they wish, and the linking service will create a new link table entry for this user each time, as in Table 1. The linking process is shown pictorially in Figure 4.1.

Each PID is regarded as a secret between the linking service and the issuing identity provider and therefore must be encrypted with the public key of the recipient when being transferred between the two.

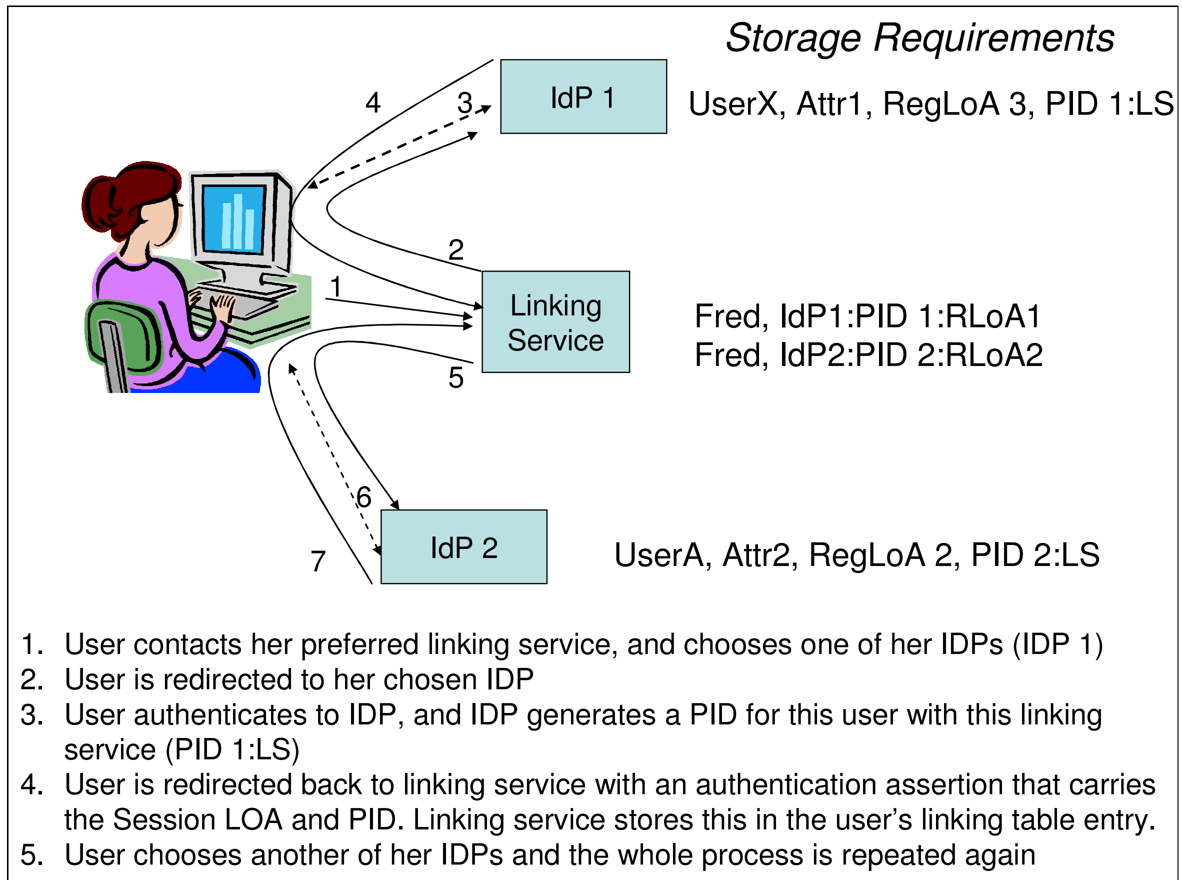


Figure 4.1. Establishing links between identity providers

4.1.2 Level of Assurance

Different identity providers will authenticate users in different ways, and to different strengths e.g. usernames and passwords are weaker than public key certificates and private keys. This is termed the Level of Authentication, or Level of Assurance (LoA). It is the assurance that a relying party can have that the user is really who it thinks he or she is. The assurance a relying party has, depends not only on the electronic authentication method that was used, but also on the initial registration process that preceded this (called Identity Proofing by NIST in [4]) for example, registering electronically over the web is much weaker than turning up in person with your passport. The US National Institute of Science and Technology (NIST) has a recommendation for LoA which classifies it in four levels, with level 4 being the strongest and level 1 being the weakest [4]. Some service providers may wish to grant a user different permissions based on the LoA of their current session. For example, if the user authenticates with an LoA of 1, they can only read the resource, but with an LoA of 3 they can modify its contents. One of the limitations of

the NIST recommendation is that the LoA is a compound metric dependent upon both the strength of the registration process and the strength of the electronic authentication method. We believe it is more useful if they are separate metrics, as described below.

Prior to any computer based authentication taking place, a user needs to register with a service, and provide various credentials to prove their identity. For example, before a new student is registered to use the University of Kent's computing services, they must first present their passport and existing qualifications, to prove they are entitled to register as a university student. We call this the *Registration LoA*. Different systems will require different registration documents and have different registration procedures, and will therefore have different Registration LoAs. After successful registration, the university allocates the student a login ID (their identifier) and associates various attributes with this in its database, e.g. degree course, student's name, date of birth, email address, department, tutor and so on. The university may offer different authentication mechanisms for student login, such as un/pw with Kerberos, un/pw with SSL, one time passwords via a mobile phone etc. Each of these mechanisms is assigned an *Authentication LoA*. When a user logs in for a session, they are assigned a *Session LoA* that equates to the Authentication LoA of the authentication mechanism they chose to use, but with one proviso. No Session LoA can be higher than the Registration LoA when attributes are going to be asserted, since it is the latter that provided the strength of authentication of the user to which the identity attributes are now attached. If no attributes are to be asserted, then the Authentication LoA can be used as is.

Returning now to the linking service, we have made provisions to include the LoA in our protocol messages. When the linking service redirects the user to an identity provider during the link registration phase, the user authenticates to the identity provider with their preferred authentication mechanism, and this has an associated Authentication LoA. The identity provider may return this as the current Session LoA to the linking service, along with the permanent identifier. The linking service stores this Session LoA as the Registration LoA of the user for this permanent identifier/identity provider tuple, as shown in Table 1.

Local User ID	PID	IDP	Registration LoA
Fred	A=12345	airmiles.com	1
Fred	EduPersonID=u23@kent.ac.uk	kent.ac.uk	2
Fred	PID=4567890	XYX.co.uk	1
Fred	UID=qwertyuiop	cardbank.com	3
Etc……			

TABLE 1. The Identity Provider Linking Table

4.1.3 Link Release Policy

The user will only wish to send different sets of linked attributes to different service providers. She will not wish to send all her linked attributes to all service providers. Consequently the user will need to create an identity provider link release policy. This tells the linking service which linked identity providers should be released to which service providers. In the simplest case, the user might indicate that all linked identity providers can be released to all service providers. This will normally be the default policy for each linking service (and is indicated by an * in each of the columns of the identity provider link release policy table, see Table 2).

In the most complex case, the user will require a different set of linked identity providers to be used with each service provider. An example of such a policy for the user known locally to the linking service as Fred is shown in table 2. This policy indicates that books.co.uk can receive attributes from three identity providers (airmiles.com, kent.ac.uk and cardbank.com); cardbank.com can receive attributes from all linked identity providers; and any other service provider should only receive attributes from the permanent identity EduPersonID=u23@kent.ac.uk from kent.ac.uk. The reason that both the permanent identifier and identity provider are held in this table is because the user may have two different identities with one identity provider, and might wish to link these together in a service provider session.

Local User ID	SP	PID	IDP
Fred	books.co.uk	A=12345	airmiles.com
Fred	cardbank.com	*	*
Fred	books.co.uk	EduPersonID=u23@kent.ac.uk	kent.ac.uk
Fred	books.co.uk	UID=qwertyuiop	cardbank.com
Fred	*	EduPersonID=u23@kent.ac.uk	kent.ac.uk
Etc…			

TABLE 2. Identity Provider Link Release Policy Table

4.1.4 Service Provision Phase

When the user wishes to use a web service, they first contact the web site. The service provider does not know who the user is, so must therefore redirect the user to their identity provider for authentication. This is the IdP discovery phase, and various different designs have catered for this. In the Shibboleth model, the service provider may use a Where Are You From (WAYF) service and redirect the user to this. In CardSpace, the service provider returns the user to their CardSpace application whereupon the user picks a card which represents their chosen identity provider. In OpenID, the user's OpenID contains the name of their Identity Provider which allows the SP to redirect the user there. Liberty Alliance (LA) has a Discovery Service [10] which can be used. SAML 2.0 defines a Common Domain Cookie (CDC) which allows members of a federation

to share information via a cookie stored in the user's browser which is held under the shared domain name of the federation [47]. Either way, the user must next present his authentication credentials to the identity provider, either directly in Shibboleth, OpenID and LA, or indirectly in CardSpace, via an authentication dialogue.

The authentication dialogue needs to be enhanced when attribute aggregation is supported, by asking the user if they wish to use attribute aggregation with this service provider. In the simplest case this can be a tick box alongside the username/password screen. Or it could be a list of Linking Services which the IdP trusts and which the user has already linked with. With direct identity provider authentication, the identity provider is able to show this enhanced screen since it knows if it has already generated one or more permanent identifiers for this user with one or more linking services. With CardSpace's indirect dialogue it is more difficult. The CardSpace application could show this enhanced screen if the service provider says that it supports attribute aggregation, but in this case CardSpace does not actually know if the user has already set up links or not. (This could be achieved by the identity provider issuing a new card to the CardSpace application after it has established a permanent identifier for this user with a linking service, but this is not a particularly user friendly solution.)

If the user chooses to perform attribute aggregation, the identity provider includes one or more *referrals* in its response to the service provider. A referral in effect says "you may find additional attributes for this user at this provider". A referral in this instance points to a linking service, and includes the permanent identifier of the user encrypted to the public key of the linking service. When the service provider receives the authentication assertion containing the user's identity attributes, if these are sufficient for the requested service, then access will be granted and no linked attributes are needed. If they are not sufficient, and the service provider supports attribute aggregation, it will follow the referral(s) by forwarding it(them) to the linking service(s) along with the authentication assertion (to prove that the user has been authenticated). It sets a Boolean in the request to the linking service either telling the latter to perform the aggregation, or saying it will perform the aggregation and referrals should be returned to it.

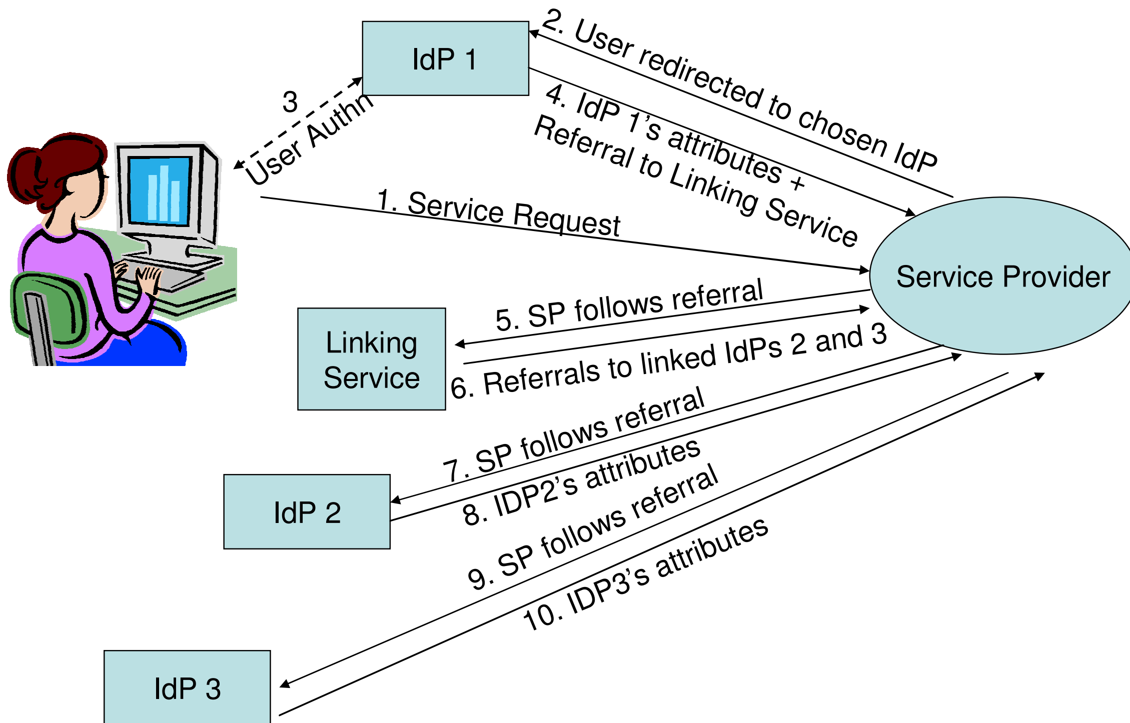


Figure 4.2. Attribute Aggregation by Service Provider

When the linking service receives the referral, it decrypts the permanent identifier and searches for this in its identity provider linking table (see Table 1). Once it has located the appropriate table entry, it retrieves the other table entries for the same user. Next it looks in its link release policy table (see Table 2) to see which of the linked identity providers can be sent to this service provider. If the service provider requested to perform the aggregation itself, then referrals to the allowed identity providers are returned, with the permanent identifiers encrypted to their respective identity providers (see Figure 4.2).

The service provider then acts on these referrals in the same way that it did with the original one. If the service provider requested the linking service to perform aggregation on its behalf, the linking service sends attribute query requests to the allowed identity providers (see Figure 4.3), forwarding the name of the service provider and the authentication assertion, so that the identity providers can encrypt their responses to the public key of the service provider and tie the attributes to the identifier found in the authentication assertion.

Finally the identity providers digitally sign their responses. In this way service providers ultimately receive an authentication assertion and multiple attribute assertions, all digitally signed by their authoritative sources, and all containing the same random identifier that the original identity provider inserted into the authentication assertion. Although the service provider does not know any of the identifiers used to uniquely identify the user at each of the identity providers, nevertheless it can be assured that the user does possess all of the identity attributes in the various attribute assertions.

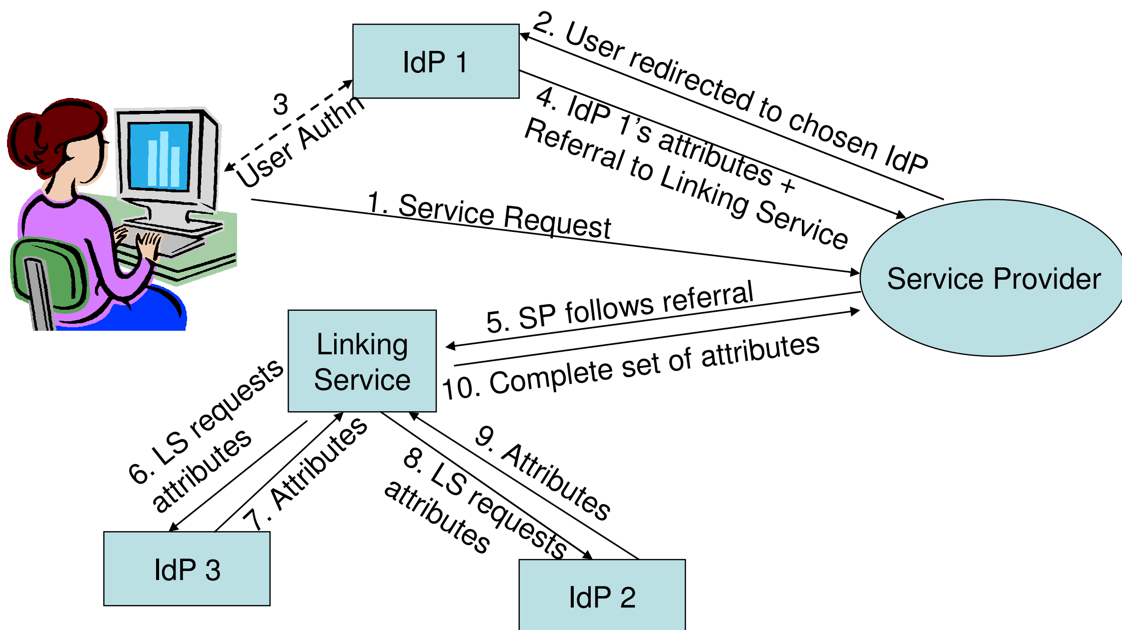


Figure 4.3. Attribute Aggregation by Linking Service

4.1.5 Using the LoA in Service Provision

The linking service may have stored Registration LoAs in its Identity Provider Linking Table during the user's link registration phase. Though not essential, they serve to improve the performance of all subsequent user-service provider sessions. During a user's service session, the linking service will only utilise linked identity providers whose Registration LoAs are higher than or equal to the current Session LoA provided by the identity provider which authenticated the

user's session. This prevents a user from creating links with low levels of assurance, and subsequently using them at higher Session LoAs, thereby pretending that the attributes have a high level of assurance. A user is allowed to create links at high Registration LoAs, and then subsequently use them on lower Session LoAs, since the service provider will know that the attributes can only be trusted up to the level of the current Session LoA.

If the linking service has stored the user's Registration LoA for a linked identity provider, and a subsequent user-service session is authenticated by a different identity provider at a lower Session LoA than this, the linking service is allowed to create a referral to the linked identity provider. The linked identity provider can then decide whether to return any attributes for the user at this low Session LoA or not. If however the subsequent user-session is authenticated by a different identity provider at a higher Session LoA than the Registration LoA, the linking service should not create a referral to the linked identity provider, since the linked identity provider should always refuse to return any attributes for the user in this high Session LoA. This is because its attributes have not been assured to such a high level and breaks the original proviso that no Authentication LoA can be higher than the original Registration LoA when attributes are being asserted.

If the linking service has not stored the user's Registration LoA for a linked identity provider, then the linking service will need to create referrals to this linked identity provider for all subsequent user-service sessions, providing it is allowed by the link release policy, and the identity provider will need to decide whether to return the user's attributes or not.

4.2 Mapping the Conceptual Model to Standard Protocols

The conceptual model has been mapped to the Security Assertions Markup Language (SAML) v2 protocol [11] during the link registration phase, and to both Liberty Alliance and CardSpace web services protocols during the service provision phase. Our attribute aggregation model provides for the passing of the LoA between the various components as part of the authentication assertion. This is based on the OASIS draft SAML profile of the NIST LoA recommendation [12]. This is not an ideal solution since it only allows us to attach a single LoA to all the assertions issued by an IdP, specifically to its authentication assertion, and no LoAs may be attached to the attribute assertions that it issues. However, until a finer granularity is required by SPs, i.e. a different LoA for each attribute an IdP asserts, it is adequate for our purposes.

4.2.1 Link Registration Protocol

The link registration protocol uses standard SAML v2.0 authentication request/response messages [11] to request user authentication by a selected identity provider and return a persistent identifier to the linking service. Upon receipt of the permanent identifier in the SAML response, the linking service will either find an existing entry in the Identity Provider Linking

Table for this permanent identifier/identity provider tuple, or a new entry will be created in the table. Either way, the user can then link additional identity provider accounts to this one.

In order to ensure that the identity provider always returns a persistent identifier to the linking service, the SAML authentication request is constrained in the following ways:

- the Format attribute of the <NameIDPolicy> element is set to : "urn:oasis:names:tc:SAML:2.0:nameid-format:persistent"
- the allowCreate attribute of the <NameIDPolicy> element is set to TRUE, which allows the identity provider to create a persistent identifier the first time around.

4.2.2 Service Provision Protocols

We have devised two possible protocol mappings for attribute aggregation using Liberty Alliance protocols, and one using CardSpace protocols. All three mappings encode referrals as Liberty Alliance ID-WSF Endpoint References (EPRs) according to the EPR generation rules defined in Section 4.2 of [13]. Each EPR points to a linking service where the service provider can find additional attributes for the user and the <sec:Token> of each EPR contains the encrypted permanent identifier of the user.

4.2.2.1 Service Provision using Liberty Alliance Protocols

Our original mapping used the Liberty Alliance ID-WSF Identity Mapping Service [13]. It required minor enhancements to the Liberty specifications. We took this mapping to the LA working group meeting in Stockholm in July 2008 for review and comment. The feedback we obtained was that the Liberty ID-WSF Discovery Service [10] might be better, since it would need fewer enhancements to the LA specifications, and open source code for the discovery service already existed, whereas none did for the identity mapping service. We then produced a mapping onto the discovery service and at the time of writing we are implementing this.

After implementation we propose to take our results and minor enhancements back to the LA group for review and comment. The disadvantage of the discovery service mapping is that it requires two round trips between the SP or linking service and the IdP each time, the first trip being to discover the endpoint reference (EPR) of the attribute authority where the random identifier is now valid and the second to pick up the attribute assertions. In this case the linking service stores the discovery services of the various IdPs. The identity mapping service only requires one round trip each time since the identity mapping service is the one stored in the linking services tables.

The SAML authentication request message issued by the service provider, asks the identity provider to generate a random identifier for the user in the authentication response (by setting the Format attribute to "urn:oasis:names:tc:SAML:2.0:nameid-format:transient") and to return both attributes and referrals (EPRs) in the response. The SAML response consists of a single

SAML assertion which contains a SSO assertion containing three statements: an SSO authentication statement, an attribute statement containing the users attributes and an attribute statement containing the EPR(s) of the linking service(s). Once the service provider has received the SAML response it may attempt to access each of the EPR's using the discovery service protocol mapping described below.

The discovery service is used for discovering web services. We can use this to discover the user's various identity providers (and attribute authorities) that have been linked together at a linking service. After the service provider receives the initial referral(s) EPR(s) from the authenticating identity provider, it sends an ID-WSF DiscoveryQuery [10] to each linking service. The DiscoveryQuery message contains the <sec:Token> copied from the referral EPR and the initial authentication assertion in the message's SOAP header [14]. The linking service decrypts the permanent identifier, retrieves the linked identity providers. If the service provider is performing aggregation, returns an ID-WSF QueryResponse. This contains referral EPRs to the discovery services of the user's linked identity providers. The service provider then sends a DiscoveryQuery message to each identity provider's discovery service, requesting the EPR of the attribute authority of the user.

Alternatively, if the linking service is performing the aggregation, it sends the same message. The identity provider's discovery service locates the user's account by decrypting the permanent identifier, and then maps the random identifier from the authentication assertion into the user's account. The identity provider returns a QueryResponse containing the EPR of the attribute authority where the random identifier is now valid.

The service provider (or linking service) sends a standard <samlp:AttributeQuery> to the attribute authority, using the random identifier, whereupon the attribute authority returns a standard <samlp:Response>, encrypted so that only the service provider can retrieve the attributes.

4.2.2.2 Service Provision using InfoCards/CardSpace Protocols

We have devised a protocol mapping for performing attribute aggregation within the CardSpace infrastructure that requires only minor changes to the CardSpace Identity Selector client and to the WS-Trust protocol.

After the user contacts the SP and is referred back to his CardSpace Identity Selector, the latter obtains the SP's security policy using the WS MetadataExchange protocol. The user picks a card and enters his login details at the prompt. If the SP has indicated in its service metadata that it accepts referral attributes, a check box labeled "Do you want to use your linked cards in this transaction?" appears below the authentication dialog. If the user checks the box, CardSpace

attempts to get his claims using a modified WS-Trust message that contains a new Boolean attribute, aggregateIdentities, which states that referrals should be returned along with the user's attributes.

Assuming the user's authentication credentials are correct, the IdP returns a CardSpace "request security token response" message that contains a single SAML SSO assertion containing an authentication statement, a SAML attribute statement containing the user's attributes, and, if the user has linked this IdP to one or more linking services, an additional SAML attribute statement containing referrals to the linking services. CardSpace relays this assertion to the SP, which utilizes these referrals to perform attribute aggregation using the Liberty Alliance discovery protocol described above.

5 Multiple Policy Authorization Evaluation Infrastructure

An authorisation infrastructure will have many different security policies such as delegation policies, access control policies, role mapping policies, credential validation policies, credential issuing policies and privacy policies. The policies may be grouped together in different ways depending upon the functional components of the infrastructure, since each functional component needs its own security policy to control how it functions. In general, security officers should be responsible for setting these security policies. Privacy policies may also be set by either the PII subject, PII keeper or PII authoritative source (this is discussed further in section 5.3 below). In order to ensure their integrity, policies should be signed by either their authors or a source that is trusted to sign on the author's behalf. Whilst system administrators are responsible for configuring systems and can thereby alter their security behaviour, by mis- or mal-administration they should only be able to cause a denial of service (DOS) attack and not an escalation of privileges. By way of example, saying which authorisation credentials are required is a policy issue and a task for the security officer, whilst configuring the details of the server to fetch the credentials from is the task of the system administrator; defining which roles to use and any role mappings to be carried out are policy issues and the task for the security officer, whilst specifying which protocol ports to use and any port mappings to be carried out can be a system administrator function. Some functions which system administrators typically carry out today should be more properly the task of the security officer since they can lead to an escalation of privileges and system compromise. For example, configuring the PKI root certificates (trust anchors) into a system should be performed (or at least validated) by the security officer, since the wrong configuration of these can lead to untrustworthy signers becoming trusted. Similarly choosing between https and http should be the task of the security officer.

We cannot assume that every security officer of every service provider and every user will use the same policy language for specifying their policy rules, or use the same policy decision point (PDP) for evaluating their policies. Today we have many examples of different policy languages e.g. XACMLv2 [15], XACMLv3 (in production), PERMIS [28], P3P [29], RT [30] etc. and many different implementations of PDPs. Consequently we need an authorization infrastructure that is capable of evaluating multiple policies written in multiple languages, and supporting multiple PDPs. In order to achieve this, the multiple policy authorization evaluation infrastructure introduces a new conceptual component called a Master PDP.

5.1 The Master PDP

The Master PDP is responsible for calling the multiple PDPs of the TAS^3 infrastructure, obtaining their authorization decisions, and then resolving any conflicts between these decisions, before returning the overall authorization decision and any resulting obligations to the AIPEP.

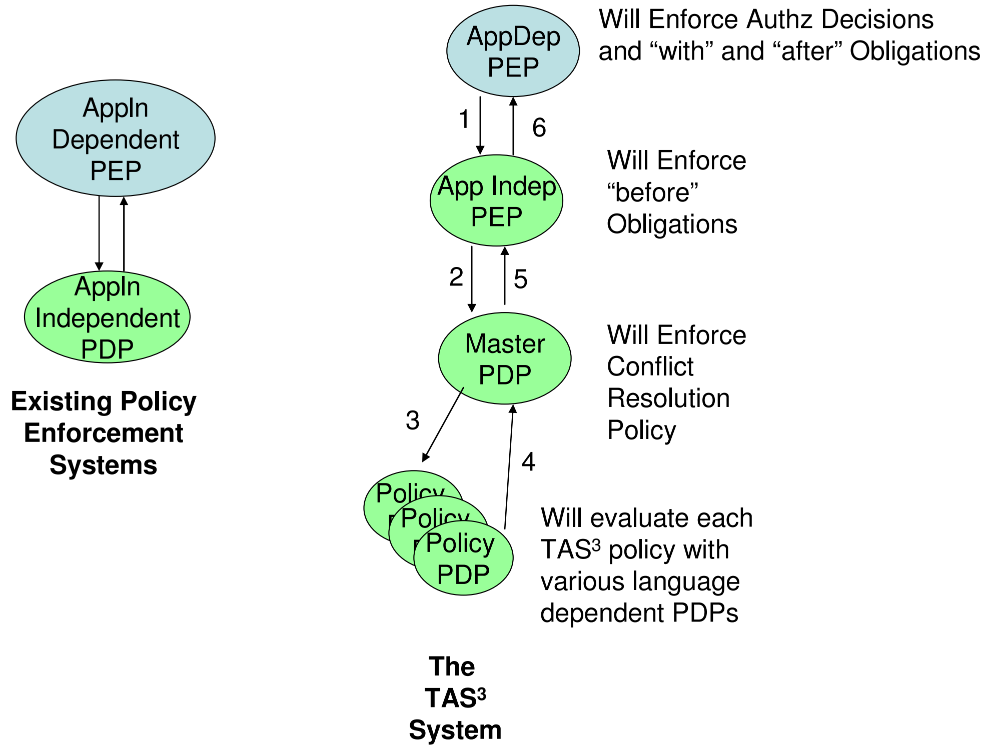


Figure 5.1. The TAS³ PEP-PDP Infrastructure

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
   xmlns:tas3="urn:?:?:tas3:schema"
   targetNamespace="tas3:to:be:decided:namespace"
   xmlns:saml="urn:oasis:names:tc:SAML:2.0:assertion"
   xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
<xs:import namespace="urn:oasis:names:tc:SAML:2.0:assertion"
   schemaLocation="http://docs.oasis-open.org/security/saml/v2.0/saml-schema-assertion-
2.0.xsd"/>
   <!-- Use a schema on the local file store as there are problems with the ones available on
the net. -->
<xs:import namespace="http://www.w3.org/2000/09/xmldsig#"
      schemaLocation="http://www.w3.org/TR/2002/REC-xmldsig-core-20020212/xmldsig-
core-schema.xsd"/>

<xs:element name="StickyPad" type="StickyPADType"/>
<xs:complexType name="StickyPADType">
   <xs:annotation>
      <xs:documentation>
         This is the TAS3 Sticky Policy and Data/resource type definition.
         The DataResource can be any data or resource which requires a sticky policy.
         Any number of policies can be stuck to a DataResource.
         The identity of the sticky policy issuer is mandatory (though could be pseudonym).
         The identity of the data subject is optional. This is because the sticky policy need not
         be stuck to a PII resource, e.g. it could be stuck to a computer system or an email
         message.
         The XML signature is optional because applications may choose to bind the PAD
         using alternate means, e.g. SSL/TLS, application protocol, SOAP envelope etc.
      </xs:documentation>
   </xs:annotation>
   <xs:sequence>
         <xs:element name="DataSubject" type="saml:SubjectType" minOccurs="0">
         </xs:element>
         <xs:element ref="saml:Issuer"/>
         <xs:element ref="StickyPolicy" maxOccurs="unbounded"/>
         <xs:element ref="ds:Signature" minOccurs="0"/>
   </xs:sequence>
</xs:complexType>
```

Figure 5.2. Sticky PAD Schema (part 1)

```
<xs:element name="StickyPolicy" type="StickyPolicyType"/>
<xs:complexType name="StickyPolicyType">
   <xs:annotation>
      <xs:documentation>
         The PolicyLanguage specifies the language the policy is written in.
         The PolicyAuthor specifies the person who wrote the policy.
         The PolicyContents contains the policy (written in the specified in PolicyLanguage).
         The PolicyType attribute specifies what type of policy this is.
      </xs:documentation>
   </xs:annotation>
   <xs:sequence>
      <xs:element name="PolicyLanguage" type="xs:anyURI"/>
      <xs:element name="PolicyAuthor" type="saml:NameIDType"/>
      <xs:element ref="PolicyContents"/>
   </xs:sequence>
   <xs:attribute name="PolicyType" type="xs:anyURI" use="required"/>
</xs:complexType>

<xs:element name="PolicyContents" type="AnyType"/>
<xs:complexType name="AnyType" mixed="true">
   <xs:sequence>
<xs:any minOccurs="0" maxOccurs="unbounded" namespace="##any" processContents="lax">
         <xs:annotation>
            <xs:documentation>
               Any content is allowed in this element.
            </xs:documentation>
         </xs:annotation>
      </xs:any>
      <xs:anyAttribute namespace="##any" processContents="lax"/>
   </xs:sequence>
</xs:complexType>

</xs:schema>
```

Figure 5.2 (cont). Sticky PAD Schema (part 2)

The Master PDP is configured with a Conflict Resolution Policy which enables it to determine which authorization decision from which policy/PDP combination should take precedence. Each of the policy PDPs will support the same interface, which in the first phase of TAS[3] will be the XACML request-response context [15]. This allows the Master PDP to call any number of PDPs, each configured with their own policy, which may be written in a different policy language to those of the other PDPs. For example, WP5 is building a behavioral trust engine which will return an authorization decision about whether the requester is trusted or not to perform the requested action. The policy language for specifying the behavioral trust rules is SWI-Prolog and is still to be finalized, but the proposed design isolated this policy language from the rest of the

authorization infrastructure, and the Master PDP will not be affected by any changes in this policy language as it evolves. The Master PDP is configured with the name of each policy language that each subordinate PDP supports, and therefore when it is passed a sticky policy (see figure 5.2) it knows which PDP to give each component of the sticky policy to.

When components such as the AIPEP and PEP reside in different systems the protocol that is used to carry the XACML request-response context is the SAML profile of XACML as described in [34]. This protocol has a field for a policy or policy reference to be inserted, which allows a TAS[3] sticky policy to be passed along with the request-response context. The SAML profile currently assumes that the policy is an XACML policy, and so we will need to propose a minor enhancement to the protocol to allow it to carry a policy in any language or languages that the PDP supports.

Consider an agency that wishes to retrieve some PII of a subject from a service provider in some country. The subject will have his own privacy protection policy written in some language e.g. P3P or XACML, which will be stored with the PII. The SP will have its own policy for who can access what information it is storing. The legislature for the country the data is held in may have its own privacy protection policy that the SP must enforce. Finally the agency will have a behavioral track record which indicates how trustworthy it is at enforcing privacy protection policies. The proposed TAS[3] design caters for this scenario by allowing all these policies and PDPs to be consulted before the agency is granted permission to retrieve the PII.

A key component of this design is the conflict resolution policy that the Master PDP will enforce. Our current thinking is that the authoritative source for each element of PII (which might be as small as an individual attribute) should be the Source of Authority (SOA) for the conflict resolution policy that is enforced by the Master PDP for that element of PII. For example, for the favourite drink attribute, the user would be the authoritative source and therefore the author of the conflict resolution policy for this element, whereas for a subject's criminal record the legal system would be the authoritative source of the information and of the conflict resolution policy for this element (since clearly the user may wish to deny everyone access to his criminal record data.) The precise contents and syntax of the conflict resolution policy is still to be determined, and is for further study. However, the XACML policy combining algorithms should form a solid base from which to build.

5.2 Sticky Policy Contents

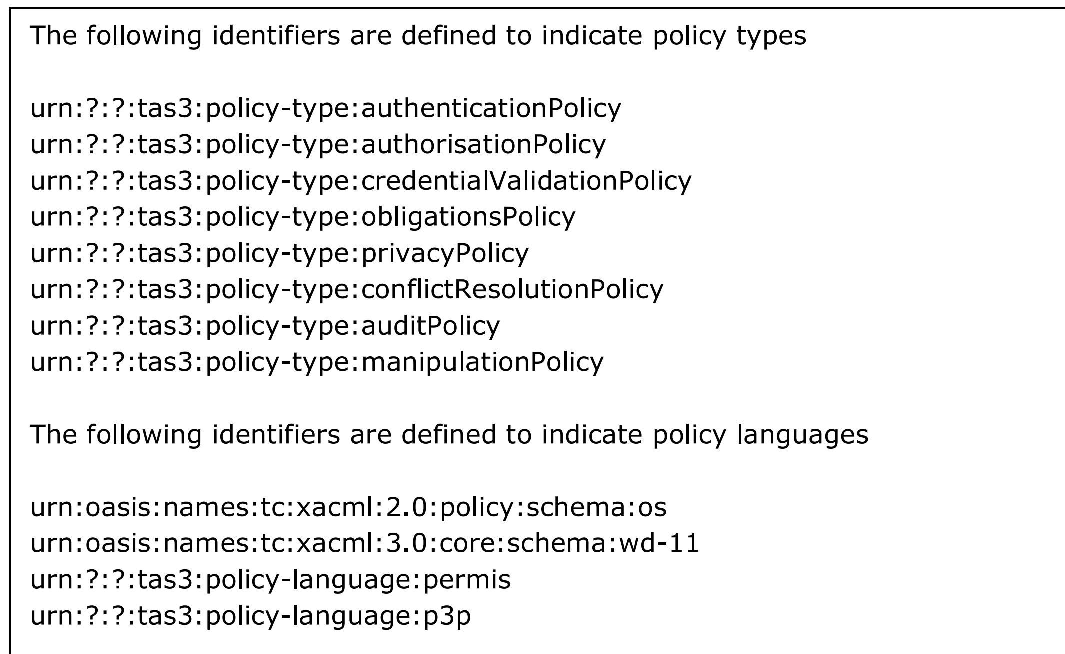
Figure 5.2 provides a schema for a (set of) **sticky p**olicy(ies) **a**nd a **d**ata/resource element combined together to create a StickyPAD. The data/resource and policies are stuck together by their source who digitally signs the combined elements. The digital signature could be the XML signature defined in the StickyPAD or an externally provided one e.g. by using SSL/TLS to transfer it. The creation of a StickyPAD may also be done in an application specific way, providing it contains the necessary data elements. For example an S/MIME message created by a

message originator could form a StickyPAD. Figure 5.2 simply provides one way using XML data elements and an XML digital signature. It is the responsibility of the sending PEP to create a StickyPAD and the receiving PEP to validate its signature when it receives the StickyPAD message in step 7 of figure 2.1. The PEP is then able to parse and unpack the StickyPAD message and re-package the various elements in the standard format of [25] ready for passing to the AIPEP.

Each embedded sticky policy in the StickyPAD is flagged with its policy language and its author. Various types of sticky policy are provisionally defined:

- authorization policy – this says who is authorized to perform which actions on the associated data/resource. There may be several of these policies in a single StickyPAD, with each policy being written by different stakeholders such as the data subject, the authoritative source of the data and the legislative authority. Each authz policy has an author, so that it can be referred to by the conflict resolution policy.
- conflict resolution policy – says how conflicts between the different authorization policy decisions are to be resolved and which authorisation decision and obligations should be returned to the AIPEP. It may contain additional obligations that are to be returned along with any obligations returned by the subordinate PDPs. There can only be one conflict resolution policy in any given StickyPAD and it must be written by the authoritative source i.e. the author of the conflict resolution policy must be the issuer of the StickyPAD.
- Credential validation policy – says how credentials have to be validated and external valid attributes mapped into internally recognized ones suitable for passing to an access control PDP
- audit policy – says what information should be audited when the associated data/resource is accessed. There may be more than one audit policy written by different authors, and the final audit policy used by the audit system will be the union of all individual audit policies contained in the sticky policy.
- obligations policies – say what actions need to be undertaken by the host system when it initially receives the StickyPAD from a remote system.
- privacy policies – contain privacy specific rules such as retention periods and purposes of use. (Note. These may be combined in the authorization policies depending upon the specific authz policy language used, but not all authorization policy languages can support all privacy specific rules). There may be more than one privacy policy in a StickyPAD and the final privacy policy will be the intersection of all the individual privacy policies.
- authentication policy – says what level of authentication/assurance (LoA) is required of requesting subjects who are to be allowed to access the associated data. (Note, whilst it is possible to represent this policy in the authorization policy through the use of Level of Assurance, as for example as described in [35], by keeping this as a separate policy it allows the PEP to short circuit the whole authorization process if the requesting subject has not been authenticated sufficiently.)
- data manipulation policy – provides rules for how the associated data (usually PII) can be transformed, enriched or aggregated with other personal data of the same data subject or of other data subjects.

Figure 5.3 shows the initial allocation of URIs to the various policy types and policy languages.



The following identifiers are defined to indicate policy types

urn:?:?:tas3:policy-type:authenticationPolicy
urn:?:?:tas3:policy-type:authorisationPolicy
urn:?:?:tas3:policy-type:credentialValidationPolicy
urn:?:?:tas3:policy-type:obligationsPolicy
urn:?:?:tas3:policy-type:privacyPolicy
urn:?:?:tas3:policy-type:conflictResolutionPolicy
urn:?:?:tas3:policy-type:auditPolicy
urn:?:?:tas3:policy-type:manipulationPolicy

The following identifiers are defined to indicate policy languages

urn:oasis:names:tc:xacml:2.0:policy:schema:os
urn:oasis:names:tc:xacml:3.0:core:schema:wd-11
urn:?:?:tas3:policy-language:permis
urn:?:?:tas3:policy-language:p3p

Figure 5.3. Various Policy Identifiers

5.3 Conflict Resolution Policy

The Master PDP is statically configured with a Conflict Resolution Policy which covers access to its static resources. In addition it may be dynamically given a Conflict Resolution Policy taken from a sticky policy attached to dynamic data. In addition it may be dynamically given a Conflict Resolution Policy taken from a sticky policy attached to dynamic data, and passed as part of the authorisation decision request [25, 34]. This policy must state how the decisions and obligations returned from the multiple subordinate PDPs are to be combined together to produce a single result. Who should be the author of the conflict resolution policy? For static resources it is clearly the resource owner. For dynamic PII resources the answer is more difficult. Candidates are: the PII subject, the authoritative source of the PII, and the PII keeper. When all these three entities coincide in one then the decision is easy, for example, if the resource is the favourite drink PII attribute of a data subject who is releasing this information to a third party, then the data subject would be the authoritative source and resource owner, so she should also be the author of the conflict resolution policy for this resource. The recipient of this PII, who becomes a PII data keeper, would not have the right to become the author of the conflict resolution policy. When the three entities do not coincide then the decision is more difficult to make. Our initial thoughts are that the authoritative source of the data being accessed should be the primary

candidate for the author of the conflict resolution policy. For example, the authoritative source of a data subject's criminal record PII is the legal system and this should be the author of the conflict resolution policy (since clearly the data subject may wish to deny everyone access to his criminal record data). Regardless of where the data is held, the original conflict resolution policy should remain. The Master PDP should ensure that any resulting conflicts in decisions from the subordinate PDPs are handled in accordance with the wishes of the authoritative source, as specified in its conflict resolution policy.

Concerning the policy's contents, the XACML standard has a reasonably comprehensive section describing policy combining algorithms and we take these as our starting point. XACMLv2 defines the following policy combining rules:

- Deny-overrides (both Ordered and Unordered) – A Deny result over-rides all other results, but otherwise the other results (Permit, Indeterminate and NotApplicable) may still be returned.
- Permit-overrides (both Ordered and Unordered) – A Permit result over-rides all other results, but otherwise the other results (Deny, Indeterminate and NotApplicable) may still be returned.
- First-applicable – After a policy has computed a Deny or Permit result, processing of all further policies stops and this first result is returned.
- Only-one-applicable – If from the set of policies only one policy is applicable to the request, then the result of evaluating this is returned. If however, multiple policies are applicable then Indeterminate is returned. This rule is the only one not appropriate for use by the Master PDP.

In addition, XACMLv3 defines the Deny-unless-permit and Permit-unless-deny algorithms. The purpose of these is to ensure that an Indeterminate or NotApplicable result is never returned to the PEP by the PDP.

If we now apply the XACML policy combining rules to the decisions returned from each of the subordinate PDPs instead of to decisions of the policies or policy sets inside an XACML policy, then these revised rules (bar Only-one-applicable) can form the basis for the Master PDP's conflict resolution policy. However, this set still does not allow the Master PDP to preferentially choose the authorization policy of the authoritative source. In order to allow this, we need an additional rule, specifically:

- Identified Author's policy overrides – this rule states that the authorization policy written by a particular identified author takes precedence over all other authorization policies.

The conflict resolution policy may also contain its own sets of obligations that should be returned along with the Deny or Permit result, which are a result of any conflict resolution. These obligations are in addition to those returned by the subordinate PDPs.

6 Dynamic Delegation of Credentials Infrastructure

Delegation of authority is an essential procedure in every modern business. A delegate is defined as “A person authorized to act as a representative for another; a deputy or an agent” (www.dictionary.com). Without delegation of authority (DOA), managers would soon become overloaded. DOA allows tasks to be disseminated between employees in a controlled manner. A delegate may be appointed for months, day or minutes, for one task, a series of tasks, or all tasks associated with a role. DOA needs to be fast and efficient with a minimum of disruption to others. Delegators should not need permission from their superiors for each act of delegation they undertake, otherwise their superiors would soon become overburdened with delegation requests from subordinates. Instead, a delegation policy should be in place so that delegators know when they are empowered to delegate (i.e. what and to whom) and when they are not.

The recipient (or service provider) who is asked to perform a service for a delegate should be able to independently verify that the delegate has been properly authorized to act as a representative for the delegator, before granting the request. If the delegate has not been properly authorised, the delegate’s request should be declined. The recipient will therefore enforce the delegation policy of its organization and deny service requests from unauthorized delegates.

In a computing environment there is also a need for DOA. One computer process may need to delegate to another computer process. One person may need to delegate his privileges to another person in order to allow the later to undertake the computer based tasks of the former. Similarly in a service oriented world, computer services also need the ability to delegate tasks to other services, so that the latter can perform subtasks on the former’s behalf. Service providers need to be able to verify that each service requestor is properly authorized. If the service requestor has been dynamically delegated authority by another authorized entity, service providers need to be able to verify that this was done in accordance with their delegation policy.

6.1 Requirements for Web Services Delegation Of Authority

As stated above, the first requirement is for a general purpose delegation of authority service that can delegate from any type of entity to any other type of entity.
(Requirement 1)

Secondly, we need to be able to independently name (or identify) the delegator and the delegate. It might be acceptable in person to grid job delegation that the grid job takes a name subordinate to that of the person, as happens with proxy certificates [36], but in person to person delegation and web service to web service delegation we should not have to make the delegate

assume a principal name which is the same as or subordinate to that of the delegator. For the reason of prudent accountability, if nothing more, every principal should authenticate with its own identity, and not with that of another. So delegation should be from one named entity to another, where their names (or identifiers) do not need to bear any relationship to each other. (Requirement 2)

When privacy protection is an additional requirement to the above, what we call *privacy preserving delegation*, then the delegator is not able to (or does not wish to) tell the system the name of the delegate. This is delegation by invitation, whereby the delegator gives an invitation (or bearer token) to the delegate, which entitles the bearer to either obtain a service directly, or obtain a delegation token from a delegation service which he can then use to obtain the service. In either case the delegate should authenticate to the service so that a proper audit can be taken of who the delegate was. (Requirement 10)

In order to build a scalable authorization infrastructure, we need to move towards attribute or role based access controls, where a principal is assigned one or more attributes, and the holder of a given set of attributes is given certain access rights to certain resources. In this way we can give access rights to a whole group of principals e.g. to anyone with an IEEE membership attribute, or to any member of project X, or any web service of a specific type, without needing to list all the members individually as there might be many thousands of them. (Requirement 3)

The delegation scheme will benefit from a hierarchical model for roles and attributes so that delegators can delegate a subset of their roles/attributes. With hierarchical roles and attributes, a principal with a superior role (or attribute) inherits all the permissions of the subordinate roles (or attributes), and may delegate a subordinate role rather than the most superior role he holds. For example, a project manager may be superior to a team leader who is superior to a team member who is superior to an employee. Principals should to be able to delegate any of their roles and attributes to other principals, so that the delegate may perform on their behalf only those tasks that are enabled by the delegated attributes. For example, a project manager should be able to delegate the subordinate role of team member to an employee. (Requirement 4)

All organizations need to be able to control the amount of delegation that is possible, in order to stop “wrong” delegations from being performed. For example, a project manager should not be able to delegate his age or name attributes to anyone else, nor be able to delegate the team member role to one of his children. So we need to have a Delegation Policy, and an effective enforcement mechanism that will control both the delegation process itself (is this delegator allowed to delegate these attributes to this delegate?) and the verification process by the consuming web service (is this delegate properly authorized to access this service?). (Requirement 5)

We may want very fine grained delegation, in order to delegate a specific task or permission rather than attributes or roles, because the latter usually confer a set of permissions to perform a set of tasks. (Requirement 6)

Users must not be constrained to having a PKI key pair before they can delegate to another entity. Users should be able to authenticate and prove their identity without having to possess a public key certificate. (Requirement 7)

A delegator should be able to prematurely revoke an act of delegation, without the delegation lasting for its originally intended period of time. When delegation takes place, its effect should be instantaneous. There are many reasons why premature revocation may be needed e.g. the delegator returns early from vacation or sick leave and wishes to continue in his role himself, or the delegate proves to be untrustworthy or incompetent in the delegated role, or the delegate completes the delegated tasks earlier than anticipated and their privileges should now be removed etc. (Requirement 8)

Finally, we wish to make the whole DOA system web services compliant, so that it will integrate nicely with the service oriented architectures (SOA) web services world that is the subject of the TAS3 project. (Requirement 9)

6.2 Design of a Delegation of Authority Web Service

We can utilise the existing PEP/PDP/CVS and ABAC models when creating a delegation of authority (DOA) web service – see Figure 6.1. The DOA web service will receive a delegation request from a delegator to delegate an attribute or attributes to a delegate. The delegated attributes are issued in the form of an attribute certificate (AC). An AC is a digitally signed attribute assertion that states that the holder (the delegate) has been assigned this set of attributes by the issuer (i.e. the delegator). This provides the DOA web service with a secure cryptographic record of the delegations. The delegator can be any web service, or a human being acting via a web services user interface. The delegate can be another web service or another human being. In this way we achieve the desired objective of person to person, service to service, person to service and service to person delegation of authority (Requirement 1). The target resource of the DOA web service, which acts as a conventional PEP, is the software that is able to issue the AC for the delegate, on behalf of the delegator. This *issue AC* software should create the attribute certificate in any standard format as required (e.g. an X.509 AC or a signed SAML attribute assertion). This *issue AC* software should have its own digital signing key pair for this task, so that future credential recipients can verify that the issued credential is authentic. Since most users do not have their own PKI key pairs they cannot issue their own ACs. This is why we require the DOA web service to sign the credential on the delegator's behalf. This solves Requirement 7.

The delegator's request will be intercepted by the DOA Web Service PEP, and passed to the CVs and PDP to ask if this user is allowed to delegate this/these particular attribute(s) to this delegate, according to the organisation's delegation and credential validation policy (Requirement 5). If the policy allows the delegator and delegate to be independently named/identified, then this solves Requirement 2. The CVS (see section 6.4)) either retrieves the delegator's current set of authorisation credentials or roles/attributes from the local repository, or they are sent by the user along with the delegation request. It validates the credentials according to its credential validation policy, then passes the valid attributes to the PDP which consults its delegation policy to see if the requested delegation is allowed or not. If attributes or roles are being delegated, the delegation policy needs to say which delegators are allowed to delegate which attributes or roles to which delegates. If very fine grained i.e. task based delegation, is to be supported, then the delegation policy also needs to say which attributes/roles are allowed to perform which tasks (Requirement 6). The PDP is then able to answer the question "is this delegator with this role/attribute allowed to delegate this task to this delegate".

If the delegator does not know or does not wish to divulge the name or identifier of the delegate (privacy preserving delegation) then authorisation will need to take place in two stages. In the first stage the PDP will need to check if the delegator is authorised to delegate the particular role/attribute/task to anyone, i.e. that the delegator does at least possess this privilege or a superset of it, and that he is allowed to delegate the privilege. The DOA web service may downgrade the request according to its own policy constraints, if the delegator does not meet all the policy requirements. If the request is accepted it will return a secret (a binary string) to the delegator along with details of any downgrade and policy constraints that apply. The delegator then passes the secret to his chosen delegate, to invite the delegate to use the service. The DOA web service will also store the request details along with the secret. Subsequently, after the delegate has authenticated itself to the DOA web service, it will present the secret to prove it is the delegate chosen by the delegator. The PDP will then check if the delegate meets any policy constraints that have been placed on the delegation, such as the delegate must be a member of the TAS^3 project.

As a result of evaluating the policy, the PDP replies granted or denied to the PEP. If granted, the PEP will ask the *Issue AC* software to issue a delegated authorisation credential to the delegate on behalf of the delegator, and will then either publish this in the local credential repository or return it to the requestor, or both. The delegate will now be able to use the issued credential to gain access to services based on the privileges that have been delegated to him, and may also be able to further delegate the embedded attribute(s) to other delegates, if allowed by the delegation policy. If the local repository stores delegated attributes instead of credentials, the *Issue AC* software will still create the delegated attribute(s) for the delegate, but will not sign them, and the delegated attribute(s) will be stored in the local repository. Subsequently the

delegate will be able to ask the DOA web service to dynamically issue a new short lived credential for him, based on the attributes that are stored for him in the local repository.

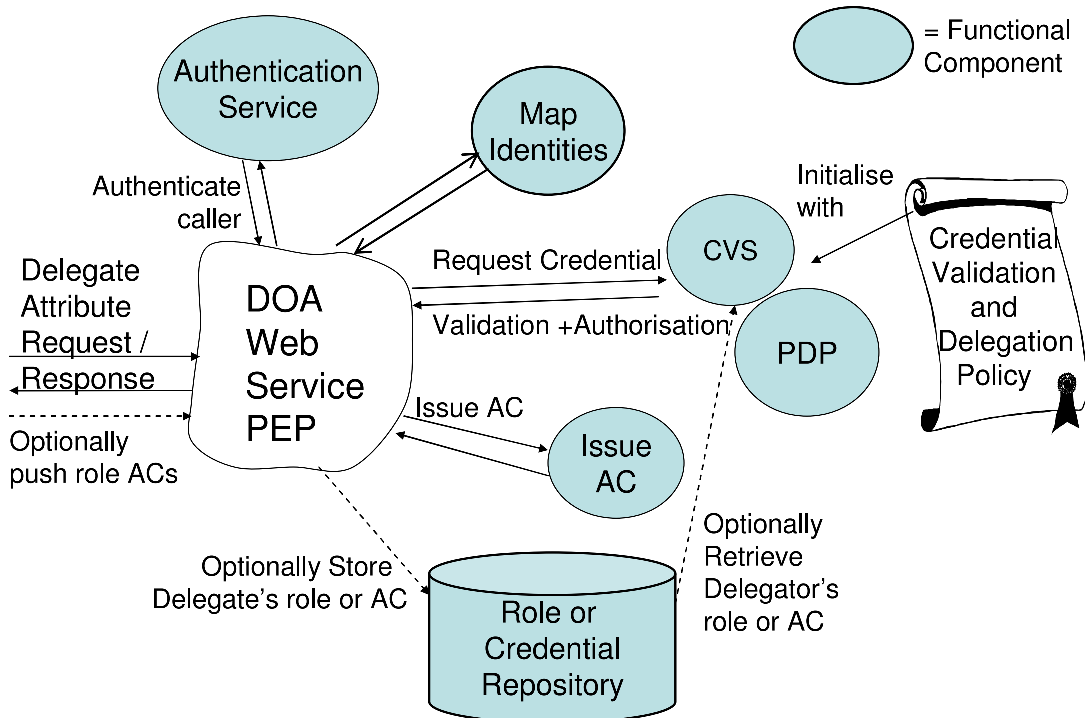


Figure 6.1. The Delegation of Authority Web Service Architecture

When a delegator makes a Delegate Attribute request to the DOA web service, the delegator is first authenticated to determine who he is. Delegator authentication can be by any suitable means, and can be via an internal authentication service or external FIM service as previously described. This model does not dictate any particular authentication scheme (Requirement 7). It is up to an implementation to determine the most appropriate authentication mechanism to use. That being said, digital signatures would be the most appropriate and secure mechanism for web service to web service authentication, but for authenticating a human user that is accessing the DOA web service via a web services user interface, the use of a federated IdP would be most appropriate as this isolates the authentication mechanism from the DOA web service and can have the IdP assert the user's permanent identifier.

The next step is to optionally map the requestor's authenticated name into the authorisation name that is held in the authorisation credentials. This step is only needed if the two names are different, for example, when proxying is used[4] or when the authentication mechanism uses a different name form to that stored in the issued credentials[5]. Ideally this step should not be needed in the latter case, since the authenticated name should be held in the authorisation credential. If the mapping is needed, how this is performed is not part of the model, but care will be needed since a security vulnerability will be introduced if the mapping is not made in a secure manner.

Once the PEP has the delegator's authorisation name, it asks the CVS to validate the delegator's credentials and then the PDP to check if this user is allowed to delegate this/these particular attribute(s) to the delegate (or to anyone in the case of privacy protected delegation). If granted is returned, the PEP then asks the target resource (*Issue AC*) to issue the new authorisation credential to the delegate, on behalf of the delegator. In the case of privacy protected delegation, the AC is issued to a secret name which represents the unknown delegate. It then stores the new credential in the repository and/or returns it to the requestor. In the case of an unknown delegate, the delegator is given the secret which he passes onto his chosen delegate. The delegate must now authenticate to the service, present the secret, and if authorised to receive the delegation token, the Issue AC service replaces the secret name with the authenticated name/identifier of the delegate.

If the delegate wishes to further delegate this credential to someone else, then the delegate will now take on the role of delegator and access the DOA web service to request delegation of this/these attribute(s) to someone or something else. In this way, delegation can continue automatically from one user to another, providing of course that each delegation is in accordance with the organisation's delegation policy.

The model supports three different modes of operation, depending upon whether the repository stores credentials (ACs) or plain attributes/roles or whether the delegator pushes authorisation credentials to the service and receives delegated credentials in return. In all cases, delegation only takes places once, but credential issuing may take place zero, one or more times. When the repository stores credentials, they are only issued once by the DOA web service, they

---

[4] For example a user may authenticate to a proxy and the proxy authenticates to the DOA web service, passing the user's name as a parameter. The mapping function would need to retrieve the user's name from the message passed by the proxy.

[5] For example, the authentication service uses usernames and passwords which are stored in the LDAP entries of the users, whilst the LDAP distinguished names are used in the authorisation credentials.

will typically have a relatively long lifetime (the period of the delegation), and they can be retrieved at will from the repository by the delegate or by web services that wish to validate the authority of the delegate to access their services. When the repository stores attributes/roles, the DOA web service can be called repeatedly by the delegate to ask it to issue typically short lived credentials based on the attributes/roles that have been delegated and stored in the repository. This service could also be used by other web services which the delegate is attempting to access, in order to retrieve the delegate's credentials. When the DOA web service is only issuing already delegated attributes, the delegator's name is not required, only the name of the delegate. But the requestor must be authenticated and authorised to ensure they are entitled to retrieve the delegate's short lived credentials. In both of these modes of operation the repository will need to record the validity period of the delegation and any policy conditions that are attached to it. If credentials are stored, this information is embedded in the issued credentials, if attributes are stored, separate fields will be needed in the repository to record it. When the repository stores attributes, it has to be strongly secured to prevent tampering with its contents and an attacker inserting false attributes. When the repository stores credentials, since the latter are digitally signed, it is not possible for an attacker to insert false credentials into the repository without first gaining access to the private signing key of the *Issue AC* service. Even if the repository is only weakly protected, the worst an attacker could do would be to remove a user's credentials – a denial of service attack. When the delegator presents his credentials during the request for delegation, and asks for the delegate's credentials to be returned to it, no repository is needed by the DOA web service. The credentials are stored external to the service, but in this case the DOA web service needs a Credential Validation Service (CVS) to validate the presented credentials. The CVS is described next.

6.3 Implementing the Delegation of Authority Web Service

In our implementation, which we term the Delegation Issuing Service (DIS), we will support the issuing of delegated credentials as either signed SAML assertions or signed X.509 ACs. SAML assertions will be short lived and non-revocable, whilst X.509 ACs may be long or short lived, and may be revocable. We will only store long lived credentials in the attached credential repository, which will be an LDAP directory. By using X.509 ACs as the long lived storage format, we protect the credentials from being tampered with and altered. However, a delegate may request that a short lived SAML assertion be created based on his long lived stored credential. The University of Kent already has a Delegation Issuing Service which is described in [16], and this will be used as the background technology that needs to be enhanced for TAS3.

For privacy preserving delegation, the DIS will validate that the delegator is allowed to delegate the authorisation to anyone, and will then store the delegated AC in a new LDAP entry created in a separate subtree (the default subtree will be beneath the DIS's own entry) using a holder name of CN=<returned secret>, <DN of subtree root>.The <DN of subtree root> is an

optional configuration parameters and is used in the secret DN so that the entries can be held in a single subtree in a reserved part of the tree. After receiving the secret the delegator will pass the secret to the delegate. The delegate will login/authenticate to the DIS, using either an IdP known to the DIS, or by registering a new account with the DIS, or using a SSL certificate. The delegate will then ask the DIS for his/her AC presenting the secret to the DIS. The DIS will check if an entry exists with the secret DN (CN=<secret>, <DN of subtree root>. If the entry is found the DIS will check if the delegate matches the constraints (e.g. has the correct roles or uses the correct IdP) and is therefore allowed to receive the AC. If so, it will issue the AC using the DN of the delegate provided at registration or login time. If not it will reject the request. The holderDN of the issued AC will be taken from either the delegate's SSL certificate, or the permanent ID provided by the IdP or the DN given at registration time. The entry with the secret DN will be deleted from LDAP.

In order to support the delegation of specific tasks rather than a role (which encompasses a set of tasks), two new "reserved" LDAP attribute types will be defined: "action" and "target". Values of these will be set by the delegator in the delegation request, instead of setting role attributes. The task based delegation will take place as follows. The DIS will call the CVS to validate the roles of the delegator. It will then call the CVS with these roles to see if the delegate is allowed these attributes (allowing downgrading of the roles). Finally the DIS will make a new call to the PDP passing the downgraded roles and the values from the action and target attributes to see if a user with these roles is allowed to perform this action on this task. If the result is granted the DIS will know that the delegator is allowed to delegate a subset of his roles to the delegate and perform this delegated action on this target and therefore it is allowed to delegate the task to the delegate.

In order to support the issuing of multiple short lived SAML assertions from a single act of delegation, the delegator will ask the DIS to delegate a role to the delegate and store it, using existing methods. The DIS will issue and store the delegated credential in its local trusted LDAP in an X.509 AC. When the Delegate asks for a short lived credential the DIS reads the LDAP entry of the Delegate, sees that Delegate has one or more long lived credentials stored, so it calls the CVS to validate them. From the set of returned valid attributes it issues a short lived credential which is a subset of the long lived stored ones. The short lived credential will always be returned to the requestor, and never stored (there is no point, since any RP that can access the credential store for the short lived credential can already retrieve the long lived credentials).

In order to support role based delegation, this will require a change to the PERMIS Delegation Policy to support role based delegation rules e.g. Project Managers can delegate the role Fire Officer to Members of Staff. The delegator (say CN=Fred, who is a project manager) asks the DIS to delegate a role (say fire officer) to someone (say CN=Mary), The DIS then gets the roles of the delegator and delegate from the CVS and checks if the role of the delegator is entitled to delegate the requested role to the role of the delegate. If allowed then the role is assigned to the

delegate by the DIS. In this case the DIS should be given all the superior roles so that when the issued credential is validated the DIS will conform to the delegation rule as it will have the required role.

6.4 Design of a Credential Validation Service

We propose a conceptual component called a Credential Validation Service (CVS). Its purpose is to validate the subject's set of credentials which are issued by multiple credential issuing services (CISs) located in different domains. A CIS is typically provided by an IdP or AA, The CVS will discard invalid (i.e. untrustworthy) attributes from these credentials and will map the valid remotely asserted attributes into locally defined ones, according to its local credential validation policy (CVP). The CVS returns a set of locally valid attributes that are suitable for presenting to the PDP.

Whilst discussing credential validation, we need to differentiate between authentic credentials and valid credentials.

- Authentic credentials are ones that have not been tampered with and are received exactly as issued by the credential issuing service. Usually a digital signature is used to prove their authenticity.
- Valid credentials are ones that are trusted for use by the recipient (i.e. the relying party).

The following examples show the difference between authentic and valid credentials:

- Example 1: Monopoly money is authentic if obtained from the Monopoly game pack. It was issued by the makers of the game of Monopoly. Monopoly money is valid for buying houses on Mayfair in the game of Monopoly, but it is not valid for buying groceries in supermarkets such as Tesco's or LIDL. However it is still authentic.
- Example 2: My credit card is an authentic credential. I can use it to buy groceries in Tesco, so it is valid there, but I cannot use it in LIDL as they do not accept credit cards. It is not valid there, but it is still authentic.

The difference between an authentic and a valid credential is whether the relying party is willing to trust the issuer of the credential to issue that particular credential for gaining access to its resources. The rules for credential validation are provided in a credential validation policy.

The XACML model has the concept of a PIP (Policy Information Point) whose purpose is to "act as a source of ***attribute*** values" [15]. The CVS is therefore a special type of XACML PIP, whose purpose is validate credentials, map their attributes into locally specified ones and return the valid attributes to the caller (PEP or PDP). There are several reasons for making the CVS a separate component to the PDP. Firstly, its purpose is to perform a distinct function from the PDP. The purpose of the (XACML) PDP is to answer the question "given this access control policy, and this subject (with this set of valid attributes), does it have the right to perform this action (with this set of attributes) on this target (with this set of attributes)" to which the answer is

essentially a Boolean, Yes or No[6]. The purpose of the CVS on the other hand is to perform the following "given this credential validation policy, and this set of (possibly delegated) credentials, please return the set of locally valid attributes for this entity" to which the answer will be a subset of the attributes in the presented credentials, possibly mapped into locally known and trusted attributes. When architecting a solution there are several things we need to do. Firstly we need a trust model that will tell the CVS which credential issuers and policy issuers to trust. Secondly we need to define a credential validation policy that will control the trust evaluation of the credentials, including mapping the validated attributes into locally known attributes. Finally we need to define the functional components that comprise the CVS.

6.4.1 The Trust Model

The CVS needs to be provided with a trusted master credential validation policy (CVP)[7]. We assume that this credential validation policy will be provided by the Policy Administration Point (PAP), which is the conceptual entity from the XACML specification that is responsible for creating policies [15]. If the PAP is trusted and there is a trusted communications channel between the PAP and the CVS, then the policy can be provided to the CVS through this channel without protection. If the channel or PAP is not trusted, or the policy is stored in an intermediate repository, then the policy should be digitally signed by a trusted policy author, typically a security officer, and the CVS configured with the public key (or distinguished name if X.509 certificates are being used) of the policy author. In addition, if the PAP or repository has several different credential validation policies available to it, which are designed to be used at different times and under different conditions, then the CVS needs to be told which policy to use. In this way the CVS can be assured of being configured with the correct credential validation policy. All other information about which sub policies, credential issuers and their respective policies to trust can be written into this master credential validation policy by the policy author.

In a distributed environment we will have many CISs each with their own issuing policies provided by their own PAPs. (A credential issuing policy (CIP) provides a CIS with the rules it should comply with when issuing credentials.) If the CVP author decides that his CVS will abide by these issuing policies there needs to be a way of securely obtaining them. Possible ways are that the CVS could be given read access to the remote PAPs, or the remote issuing authorities

[6] XACML also supports other answers: indeterminate (meaning an error) and not applicable (meaning no applicable policy), but these are conceptually other forms of No.
[7] Note that whilst we refer to the policy in the singular, we acknowledge that it will contain multiple policy statements, and therefore may be regarded as a set of policy rules.

could be given write access to the local PAP, or more realistically, the issuing policies can be bound to their issued credentials and obtained dynamically during credential validation. Whichever way is used, the issuing policies should be digitally signed by their respective issuers so that the CVS can evaluate their authenticity. If the issuing policies are bound to the credentials, then a single signature over all the information will suffice.

The policy author may decide to completely ignore all the issuer's policies, or to use them in combination with his own credential validation policy, or to use them in place of his own policy. Thus this information (or policy combining rule) needs to be conveyed as part of the CVS's policy.

6.4.2 The Credential Validation Policy

The CVS's policy needs to comprise the following components:

- a list of trusted credential issuers. These are the issuers in the local and remote domains who are trusted to issue credentials that are valid in the local domain. They are the roots of trust. This list is needed so that the signatures on credentials and policies can be validated. The list could contain the raw public keys of the issuers or it could refer to them by their X.500 distinguished names or their X.509 public key certificates.
- the hierarchical relationships of the various sets of attributes. Some attributes, such as roles, form a natural hierarchy. Other attributes, such as file permissions might also form one e.g. *all* permissions is superior to *read, write* and *delete*; and *write* is superior to *append* and *delete*. When an attribute holder delegates a subordinate attribute to another entity, the credential validation service needs to understand the hierarchical relationship and whether the delegation is valid or not. For example, if a holder with a manager role delegates the administrator role to someone, is this a valid delegation or not? The relationship of manager to administrator in the attribute hierarchy will provide the answer to this question.
- a description (schema) of the valid delegation graph. The process of delegation forms a directed acyclic graph (DAG), with the Privilege Management Infrastructure (PMI) roots of trust as the sources of the graph. (PMI roots of trust are to authorisation what PKI roots of trust are to authentication.) Intermediate nodes in the graph represent delegates who subsequently act as delegators and further delegate their attributes (or permissions) to others. Sink nodes represent delegates who have not further delegated their attributes (or permissions) to others. Edges in the graph represent the attributes or permissions that have been delegated from the delegator to the delegate. Successor edges must always represent the same or less attributes and permissions than the union of their predecessor edges, otherwise a delegator will have delegated more privileges than he himself possessed. The graph is acyclic because a delegator should not be able to delegate to herself or to a predecessor. Rationally, there is a reason for this – a delegate should never *need* to delegate to an entity that previously delegated directly or indirectly to it. But there is also a security reason for this. There is a potential security loophole if a delegator, who is allowed to delegate a

privilege to others but not to assert it, does subsequently delegate it to herself, then she would be able to assert the delegated privilege. This CVS policy component describes how the CVS can determine if a chain of delegated credentials and/or policies falls within a trusted graph or not. This is obviously a complex policy component. One way of simplifying it, is to restrict the directed graph into being a delegation tree, in which there is only one source or PMI root node which holds all the attributes that can be delegated, and each act of delegation creates a separate delegate subordinate node. If a delegate receives attributes from two or more delegators in separate acts of delegation, then these are represented as separate edges and nodes in the tree, without merging the delegate nodes together. Delegation trees significantly simplify the process of credential validation and credential revocation because each credential only has a single parent. Even then, there is no widely accepted standard way of describing delegation trees. One approach can be found in X.509 [17] and a different approach in [18]. The essential elements however should specify who is allowed to be in the tree (both as an issuer and/or a subject), what attributes they can validly have (assert) and delegate, and what constraints apply.

- any validity constraints on the various credentials (e.g. time constraints or target constraints). The CVS's policy may place its own constraints on credential validity regardless of those of the issuer.
- the mapping rules which specify which valid remote attributes (or permissions) map into which local attributes (or permissions). If role or attribute hierarchies are used in either the remote or local domains, then the mapping algorithm will need to know what these hierarchical relationships are in order to perform the correct mapping. The alternative is to ignore the hierarchies in the remote domains and to specify individual mapping rules for every single attribute from the remote domain.
- finally, we need a disjunctive/conjunctive directive (or policy combining rule) to say how to intersect the issuer's policy with the CVS's own policy. The options are: only the issuer's issuing and delegation policy should take effect, or only the CVS's policy should take effect, or both should take effect and valid credentials must conform to both policies.

Note that when dynamic delegation of authority is not being supported, the above policy can still be used in a simplified form where a delegation tree or graph reduces to a one level hierarchy, in which the root node(s) are the set of trusted issuers and the first level are the set of users who can be issued with credentials. In this case the CVS's policy now controls which trusted issuers are allowed to assign which attributes to which subjects, along with the various constraints, attribute mappings and disjunctive/conjunctive directive.

An important requirement for multi-domain dynamic delegation is the ability to accept only part of an asserted credential. This means that the policy should be expressive enough to specify what is the maximum acceptable set of attributes that can be issued by one Issuer to a Subject, and the evaluation mechanism must be able to compute the intersection of this with those that the Subject's credential asserts. This model is based on full independence of the issuing domain

from the validating domains. In general it is impossible for a validating domain to fully accept an arbitrary set of credentials from an issuing domain, since the issuing and validating policies will not match. It is not always possible for the issuing domain to tell in advance in what context a subject's credentials will be used (unless new credentials are issued every time a subject requests access to a resource) so it is not possible to tell in advance what validation policy will be applied to them.

6.4.3 The CVS functional components

Figure 6.2 illustrates the architecture of the CVS function and the general flow of information and sequence of events. First of all the service is initialised by giving it the credential validation policy (step 0). Now the CVS can be queried for the valid local attributes of an entity (step 1). Between the request for attributes and returning them (steps 1 and 6) the following events may occur a number of times, as necessary i.e. the CVS is capable of recursively calling itself as it determines the path in a delegation tree from a given node to a PMI root of trust. The Policy Enforcer requests credentials from a Credential Provider (step 2). When operating in credential pull mode, the credentials are dynamically pulled from one or more remote credential providers (these could be SAML authorities, AA servers, LDAP repositories etc.). The actual attribute request protocol (e.g. SAML or LDAP) is handled by a Credential Retriever module. When operating in credential push mode, the CVS client stores the already obtained credentials in a local credential provider repository and pushes the repository to the CVS, so that the CVS can operate in logically the same way for both push and pull modes. When performing attribute aggregation as described in section 4, it is the Credential Retriever that pulls the credentials from the remote set of authorities. After credential retrieval, the Credential Retriever module passes the credentials to a decoding module (step 3). From here they undergo the first stage of validation – credential authentication (step 4). Because only the Credential Decoder is aware of the actual format of the credentials, it has to be responsible for authenticating the credentials using an appropriate Credential Authenticator module. Consequently, both the Credential Decoder and Credential Authenticator modules are encoding specific modules. Whilst in principle the Credential Authenticator module should not need to be encoding specific and should be able to rely on the same underlying authentication infrastructure (usually a PKI) to validate the digital signatures on the credentials, regardless of their format, in reality there currently is no standard interface for signature verification that can be called for different token formats. Hence the Credential Decoder is currently an encoding specific module. The Credential Decoder subsequently discards all credentials that are deemed by the Authenticator module to be unauthentic – these are ones whose digital signatures are invalid, either cryptography or because the signer's certificate cannot be traced to a PKI root of trust, or because the signer's certificate has been revoked. Authentic credentials on the other hand are decoded and transformed into an implementation specific local format that the Policy Enforcer is able to handle (step 5).

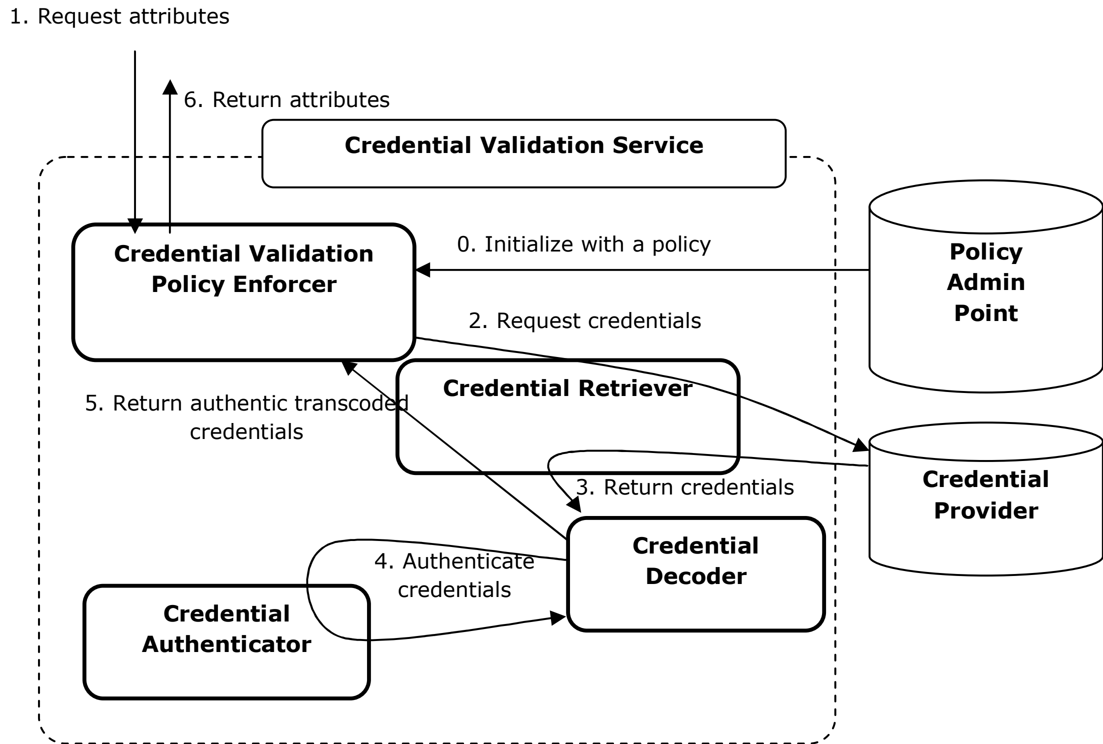


Figure 6.2. Data Flow Diagram for Credential Validation Service Architecture

The task of the Policy Enforcer is to decide if each authentic credential is valid (i.e. trusted) or not, and if it is, to extract the attributes from it and map them into locally valid ones. It does this by referring to its Credential Validation policy to see if the credential has been issued by a PMI root of trust or not. If it has, it is valid. If it has not, the Policy Enforcer has to work its way up the delegation tree (or graph) from the current credential to its issuer, and from there to its issuer, recursively, until a PMI root of trust is located, or no further issuers can be found (in which case the credential is not trusted and is discarded). Consequently steps 2-5 are recursively repeated until closure is reached. Even when the delegation graph has been simplified to a delegation tree, in the general case there will be multiple trees each with their own PMI root of trust, who each may have their own Issuing Policy, which may have been further restricted by their delegates, which may then need to be adhered to or not by the Policy Enforcer according to the CVS's policy. There are also issues of height first or breadth first upwards tree walking, or top-down vs. bottom-up tree walking. These are primarily implementation rather than conceptual issues, as they effect performance and quality of service, and so we will address these later when we describe our implementation of a CVS. Once a credential is found to be valid, its attributes are

mapped into locally valid attributes according to the attribute mapping rules, and these are returned to the caller.

The proposed architecture makes sure that the CVS can:

- Retrieve credentials from a variety of physical resources
- Aggregate credentials from different authoritative sources
- Decode the credentials from a variety of encoding formats
- Authenticate and perform integrity checks specific to the credential encoding format

All this is necessary because realistically there is no way that all of these will fully match between truly independent issuing domains and the relying party.

6.5 Implementing the initial CVS

There are a number of challenges involved in building a fully functional CVS that is flexible enough to support the multiple requirements outlined above. Firstly we need to fully specify the Credential Validation Policy, including the rules for constructing delegation graphs (or multiple trees) and attribute mappings. Then we have to engineer the policy enforcer with an appropriate algorithm that can efficiently navigate the delegation graph (or a tree) and determine whether a subject's credentials are valid or not. In our implementation we have chosen to constrain the delegation DAG into a set of delegation trees, with each tree having a single PMI root of trust. Finally we have to map the attributes from valid credentials into locally valid attributes. The output from the CVS is a set of locally valid attributes encoded in XACML format ready for passing to the PDP.

We have implemented our CVS policy in XML, according to the schema shown in Appendix 2. Most components of the policy are relatively straightforward to define, apart from the delegation trees. We have specified the list of trusted credential issuers (PMI roots of trust) which we call Sources of Authority (SoAs), by using either their subject distinguished names (DNs) or their subjectAltName Uniform Resource Identifiers (URIs) from their X.509 public key certificates. Only the latter subjectAltName is supported since this is the naming scheme used by all entities on the world wide web. We chose to use DNs or URLs rather than public keys for two reasons. Firstly, they are easier for policy writers to understand and handle, and secondly it makes the policy independent of the current key pair that happens to be in use by a trusted issuer. The authorisation policy is therefore independent of the underlying PKI, but nevertheless points to the entities from the PKI that are trusted to act as PMI roots of trust.

Multiple disjoint attribute hierarchies are supported. Each attribute hierarchy is specified by listing superior-subordinate attribute value pairs. This allows any arbitrary partial order to be created, since there is no limit to the number of times a particular attribute value can occur as

either a superior or a subordinate value in one hierarchy (subject to the restriction that loops are not created). Attributes and attribute values can be independent of any hierarchy if so wished, so that permission inheritance does not have to be supported if it is not required. The attribute hierarchies of remote domains/SoAs and the local domain/SoA can both be specified in the same way, and both can used in the attribute mapping policy.

The attribute (or role) mapping policy is specified as a set of attribute (or role) mapping rules, where each rule comprises an attribute value or role from an external domain and an attribute value or role from the local domain into which it should be mapped. When attribute hierarchies are being supported, then an external role will inherit all the privileges of any subordinate internal roles as well as the specific internal role(s) into which it is mapped. Any external role not mentioned in the role mapping policy, that is superior to an external role (or roles) which is (are) mentioned in the role hierarchy policy, will map into the same internal role(s) as the mentioned external role(s). Any non-mentioned external roles that are subordinate to mentioned external roles will be discarded as they have less privileges than the lowest mapped external roles.

Delegation trees have each been defined as a name space (a delegation domain), a delegation depth and a root of trust. Anyone in the delegation domain who is given a credential by the designated root of trust may delegate it to anyone else in the same domain, who in turn may delegate it to anyone else in the same domain until the delegation depth is reached. X.500/LDAP distinguished names or HTTP URLs are used to define the delegation domains. A base DN or URL is used to specify the root node of the delegation domain, and the domain may be refined by defining included and excluded subtrees so that any arbitrary subtree may be constructed. All delegates must belong to the refined domain otherwise the delegation is not valid. Since we already refer to the credential issuers (roots of trust) by their LDAP DNs or URLs, it was natural to refer to the delegates in a delegation tree by their DNs or URLs as well. In this way we can easily link delegation chains together by matching the issuer in one certificate with the subject in the next certificate in the chain. We recognise that a more flexible approach to defining delegation trees is by referring to delegates by their attributes rather than their DNs or URLs, as for example as used by Bandmann et al [38]. Their delegation tree model allows a policy writer to specify delegation trees such as “anyone with a head of department attribute may delegate a project manager attribute to any member of staff in the department”. This is a planned enhancement which will be carried out next year. It should be noted that this introduces a level of indirection, complexity and performance penalty, because the CVS will have to retrieve the delegate’s and delegator’s credentials, extract their attributes from these, see if they have an attribute that matches the one(s) in the delegation rule, and then validate that each attribute was correctly assigned or delegated in the credential according to its governing rule. This adds a level of complexity that our current model does not have, since in our current model we simply need to match on the delegate or delegator’s name.

One obvious constraint that we place on our delegation trees is that the same attribute value (or one of its subordinate values in the role hierarchy) must be propagated down any given tree from the root of trust, and either new unrelated attributes that are not in the same role hierarchy, or superior values from the same role hierarchy, cannot be introduced in the middle of a delegation tree. This is to ensure that a delegator can only delegate his existing permissions or a subset of them, and not an unrelated set or superset. A new delegation tree would need to be specified for the delegation of an unrelated or superior attribute. Whilst the current implementation only supports the delegation of attributes/roles, we plan to add the delegation of specific tasks/permissions to the next version.

Trusted issuers and delegation domains are defined separately in the policy and then linked together with the attributes that each issuer is trusted to issue, along with any additional time/validity constraints that are placed on the issued credentials. (The constraints have not been shown in the schema.) The reason for doing this is improved flexibility, since one trusted issuer may be the root of several delegation trees, and one delegation domain may have several roots of trust.

Our current implementation only supports delegation credentials that are X.509 attribute certificates, which we store in an LDAP directory. We plan to add support for SAML attribute assertions in the next version.

In our current implementation we do not pass the full Issuing Policy along with the issued credential, we only pass the tree *depth* integer, since this was already specified in an X.509 standard extension. Therefore the CVS does not know what the issuer's intended delegation tree is. We have assumed that the credential issuing software, which in our case is the delegation service at the issuing site, will enforce the Issuing Delegation Policy and so only credentials that conform to the Issuing Policy will be issued. However, the CVS policy writer is able to specify his own delegation domain for the received credentials and this may be more restrictive than that of the issuing domain, or the same as or less restrictive than it. So ultimately the owner of the resource will control the delegation tree that is deemed to be valid at the target site. In order to ensure that the Issuing Delegation Policy is enforced at the target site the issuer's delegation tree should be configured into the CVS's policy. This assumes that the structure of the issuer's delegation tree is the same as that of our CVS policy, which will not always be the case in independent domains using different models and software implementations. A future enhancement would be to carry the complete Issuing Delegation Policy in each issued credential, and to allow the CVS's policy writer to enforce it, or overwrite it with his own policy, or force conformance to both. In this way a more sophisticated delegation tree can be adhered to. This of course will depend upon there being a standardised format for the transfer of Issuing Delegation Policies in credentials, which currently there is not for either SAML attribute assertions or X.509 or SPKI certificates. So we propose to leave this out of the scope of the TAS^3 project.

6.5.1 Delegation Tree Navigation

Given a subject's credential, the CVS needs to create a path between it and a root of trust, or if no path can be found, conclude that the credential cannot be trusted. There are two alternative conceptual ways of creating this path, either top-down, also known as backwards [39] (i.e. start at a root of trust and work down the delegation tree to all the leaves until the subject's credentials are found) or bottom-up, also known as forwards (i.e. start with the subject's credential and work up the delegation tree until you arrive at its root of trust). Neither approach is without its difficulties. Either way can fail if all the credentials are not pushed to the CVS. If the CVS has to pull credentials from the issuers or their repositories, then all the credentials have to be held consistently – either all with their subjects or all with their issuers, otherwise the CVS will not be able to efficiently locate them. In our implementation all credentials are held with their subjects, typically in their LDAP directory entries, or more recently, in files linked to their DNs held in WebDAV repositories [40]. As Li et al point out [39], building an authorisation credential chain is more difficult in general than building an X.509 public key certificate chain, because in the latter one merely has to follow the subject/issuer chain in a tree, whereas in the former, a DAG rather than a tree may be encountered. Graphs may arise for example when a superior delegates some permissions in a single credential that have been derived from two of more credentials that he possesses, or when attribute mappings occur between different authorities. Our CVS implementation is currently limited to supporting delegation trees rather than DAGs, and so it will not follow multiple superior credentials from a single subordinate one as these are forbidden. Delegations are also restricted to occurring in a single subject domain, and therefore attribute mappings will not occur. But even for the simpler PKI certificate chains, which our credential chains conform to, there is no best direction for validating them. SPKI uses the forwards chaining approach [41]. As Elley et al describe in [42], in the X.509 model it all depends upon the PKI trust model and the number of policy related certificate extensions that are present to aid in filtering out untrusted certificates, whether backwards or forwards chaining is preferrable. Given that our delegation tree is more similar to a PKI tree, and that we do not have the policy controls to filter the top-down (backwards) approach, and furthermore, we support multiple roots of trust so in general would not know where to start, then the top-down method is not appropriate.

There are two ways of performing bottom-up (forwards) validation, either height first in which the immediately superior credential only is obtained, recursively until the root is reached, or breadth first in which all the credentials of the immediate superior are obtained, and then all the credentials of their issuers are obtained recursively until the root or roots are reached. The latter approach may seem counter-intuitive, and certainly is not sensible to perform in real time in a large scale system, however a variant of it may be necessary in certain cases, i.e. when DAGs are supported, or when a superior possesses multiple identical credentials issued by different authorities. Furthermore, given that in the TAS^3 federation model we allow a user to simply authenticate to a gateway and for the system to determine what the user is authorised to do (the

credential pull model), the first step of the credential validation process is to fetch all the credentials of the user. This is performed by the Credential Retriever in Figure 6.2. Thus if the CVS recursively calls itself, the breadth first approach would be the default credential retrieval method. Thus we have added a retrieval directive to the credential validation method, which is set to breadth first for the initial call to the CVS, and then to height first for subsequent recursive calls that the CVS makes to itself.

In order to efficiently solve the problem of finding credentials, we add a pointer in each issued credential that points to the location of the issuer's credential(s) which are superior to this one in the delegation tree. This pointer is the AuthorityInformationAccess extension defined in RFC3280 [43]. Although this pointer is not essential in limited systems that have a way of locating all the credential repositories, in the general case it is needed.

In order to ensure that a delegator does not overstep his or her authority, after retrieving the attribute(s) from the delegator's credential we need to check that one of them is superior or equal to all the attributes in the delegate's credential in the attribute hierarchy. If it is not superior to all of the delegate's attributes in the attribute hierarchy, the delegator has exceeded his authority and the delegate's credential is discarded and processing stops.

In the case of relatively long lived credentials, revocation is clearly an issue. When a credential has been revoked, which we achieve by removing it from its LDAP store, then all the credentials in the branch of the tree for which the revoked credential is the root, are also considered to be revoked. If any credential between the requestor's credential and the root of trust has been revoked, then the requestor's credential is considered to be invalid, and processing stops. We have also implemented a novel scheme for revoking credentials which uses the web as a finite state machine to indicate the revocation status of each credential [40]. This scheme inherently supports instant revocation and can be more efficient than using CRLs.

Finally, as a means of enhancing performance, we envisage two mechanisms. Firstly all credentials that are retrieved during a validation exercise can have their attributes cached locally after their validation, so that if a subsequent request requires the same credential, it will not need to be pulled and validated again. We plan to add this to the next version of the CVS, using a cache time that is either approximately equal to the period of CRL issuance (for long lived credentials) or the credential's lifetime (for short lived ones). Secondly, when long lived credentials are used, a background task could be run when the system is idle, that works its way down all the delegation trees from the roots of trust, in a breadth first search for credentials, validates them against the CVS's policy, and caches the valid attributes for the same period as before. Then when a user attempts to access a resource, the CVS will be able to give much faster responses because the high level branches of the delegation tree will have already been validated. This will not work however when short lived credentials are issued on demand, since the CVS will not be able to pull these prior to their issuance.

A fuller description of our CVS and how it can be used to add dynamic delegation of authority to XACMLv2 policies, can be found in [37].

7 Dynamic Management of Policies Infrastructure

In a web services environment, there are issuing domains that issue credentials to users and target domains that consume credentials. The authorization policy of the target domain decides whether an issued credential is to be trusted or not i.e. is valid or not, and whether it provides sufficient permissions or not to the accessed resource. In an attribute (or role) based authorisation policy, the permission-attribute assignment (PAA) rules form the access control policy. The user-role assignment (URA) rules form the credential validation policy. Thus, an authorisation policy includes an access control policy and a credential validation policy. It is set by the administrator of the target domain, in consultation with the administrators of the issuing domains. When there are multiple authorisation policies and multiple PDPs, it can be a large and daunting task for one administrator to manage. The administrator may therefore need to delegate authority to other administrators to manage (some of) the policies (requirement 1). However, the administrator needs to remain in overall control and only delegate a subset of his permissions (requirement 2). Furthermore, if the system is to be responsive to changes, policies will need to be updated dynamically without having to shut down and then restart the system (requirement 3). If the administrator has different agreements with the administrators of different issuing domains, then the rules for each of these issuing domains must be kept separate and should not be mixed up by the authorisation system (requirement 4).

Organisations assign organisational roles to individuals e.g. managing director, team leader etc. but these do not equate to the roles understood or used by workflows. However the authorisation system authorises these individuals to participate in the tasks of workflows, and therefore it would be beneficial if the authorisation system could utilise the individual's organisational roles when granting permissions to participate in the workflow (requirement 5).

In this section, we propose a dynamic management of policies model which provides the following features for authorisation administration in a web services world:

- Policies can be updated dynamically without having to shut down and restart the system. This addresses requirement 3.
- Administrative roles are defined which grant permission to dynamically update limited parts of the authorisation policy in the target domain, more specifically, to assign organizational level attributes to a subset of the privileges which grant access to a service's workflow resources (see Figure 7.1). This addresses requirement 2.
- Administrators are dynamically created by assigning these administrative roles to them. These roles can be dynamically delegated, and also dynamically revoked, thereby dynamically adding and removing administrators from the system. This addresses requirement 1.

- An administrator can dynamically assign a subset of the workflow permissions granted by the administrative role, to any organizational level user attributes (i.e. perform PAA). In addition, the administrator can provide the policy information for validating the user credentials that contain these attributes (i.e. URA validation). This addresses requirements 3 and 2.
- Collaborations between organisations are independent of each other, since an organisation's workflow privileges are independent of those of other organisations. This addresses requirement 4.
- Application-level (workflow) security infrastructures are separated from organisational level security infrastructures since workflow permissions are dynamically assigned to organizational level attributes. This addresses requirement 5.

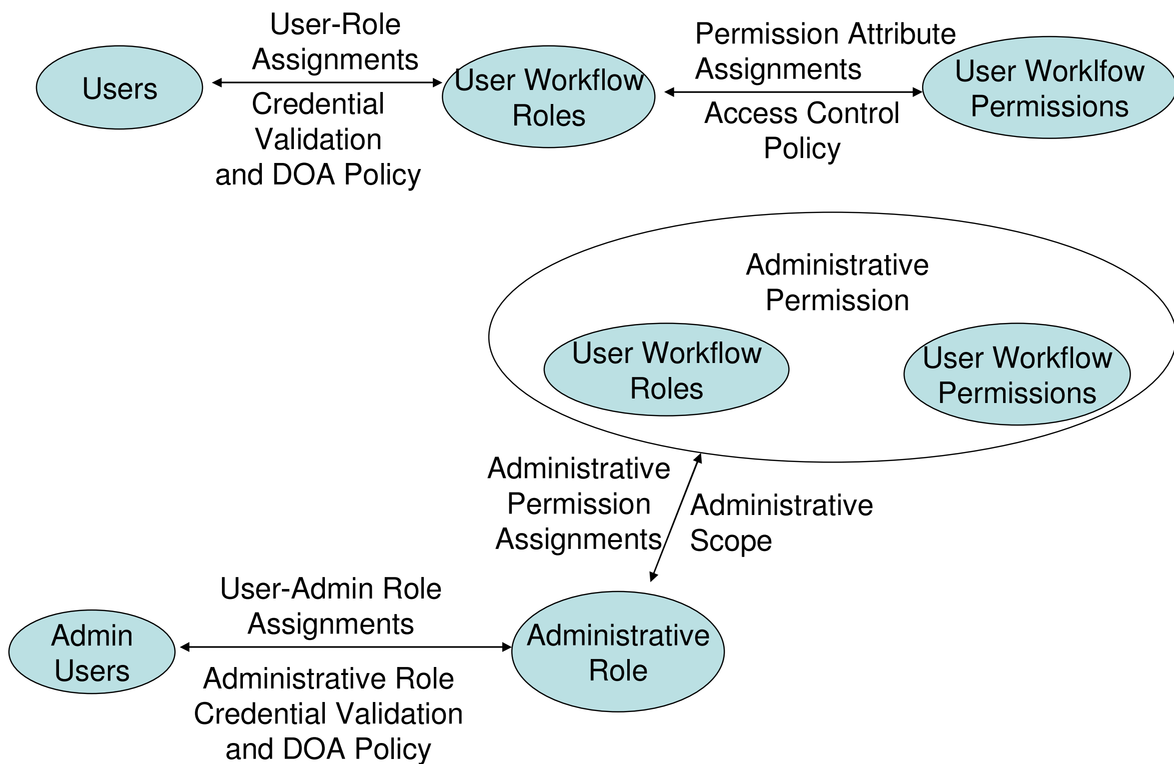


Figure 7.1. User Roles and Administrative Roles

By allowing authorization policies to be dynamically updated as above, our model allows the authorisation system of a target domain to dynamically *recognise* trusted administrators, to

dynamically *recognise* the new attributes they are trusted to issue, and to dynamically *recognise* new users of the VO. The initial definition of the administrative roles means that the authorization system knows the limit of their administrative authority in assigning permissions to users. We call this model Recognition of Authority.

There are two approaches for assigning permissions in a local organisation to users in partner organisations. The first is to directly assign workflow permissions to remote user attributes and the second is to map remote user attributes into local workflow roles/attributes by attribute-role mapping. Both approaches can facilitate collaborations between organisations. In the attribute-role mapping approach, the permissions given to a remote attribute are the workflow permissions of the local role, which is fixed. Thus, this approach limits the granularity of delegation to that of the pre-defined local workflow roles (and their subordinate roles), whilst direct permission assignment allows each workflow permission to be delegated or assigned separately. On the other hand, by mapping remote user attributes to local workflow roles, the changes of participants in a workflow are confined to the modification of mappings from an organisation's attributes to the local workflow roles (it does not affect the workflow's specification) and changes to the specification of local workflow roles do not require modifications to the remote user attribute specifications. Thus, this approach supports the separation of workflows from organisational changes. Since both approaches have their merits, our model is designed to support both approaches. When an administrative role is defined, its administrative permissions are defined as either an ability to assign a restricted set of workflow permissions to any user attributes, or an ability to map any user attributes into a restricted set of existing local workflow roles.

We identify two types of permission: a *normal permission* (or *workflow permission*) and an *administrative permission*. A workflow permission grants a user permission to perform a particular workflow action on a particular resource under certain conditions. An administrative permission grants an administrator permission to perform either PAA, or to perform role mappings to workflow roles under certain conditions.

When a set of workflow permissions is given to an attribute or role, we say that the role or attribute is a *workflow role*. When a set of administrative permissions is given to a role we say the role is an *administrative role*. Someone who holds an administrative role is called an administrator. The set of workflow permissions and workflow roles that an administrator can assign or map to new user attributes is called his *administrative scope*.

The recognition of authority management model for facilitating dynamic collaboration between organisations comprises the following steps:

1. The policy writer (SoA) of the target domain defines a set of administrative roles for the target domain, an administrative role credential validation policy, and the workflow permissions that are attached to these administrative roles (i.e. the administrative scope).

2. The SoA dynamically delegates these administrative roles to trusted people in remote domains with whom there is to be a collaboration, by issuing administrative role credentials to them.
3. To establish a collaboration, one of these administrators must update the SoA's authorisation policy by writing a collaboration policy. The collaboration policy includes an access control policy and/or a role mapping policy, and a user credential validation policy. The latter specifies validation rules for user credentials containing newly defined (organizational level) user attributes, whilst the former specifies either permission attribute assignments or role mappings for the newly defined user attributes. In this way, users who hold credentials containing these new attributes will gain access to the appropriate target workflow resources.
4. In order to ensure that no administrator can overstep his delegated authority, the authorisation system has to validate that the collaboration policy lies within the the administrative scope specified in 1. above. If it does, it is accepted, and its policy rules become dynamically incorporated into the SoA's policy. If it does not, it is rejected, and its policy rules will be ignored.
5. When a user from a collaborating domain wants to access a protected resource in the target domain, assuming the collaboration policy has been accepted, the authorisation system retrieves and validates the user's credentials/attributes against the now enlarged credential validation policy. Only valid attributes will then be used by the access control system to make access control decisions for the user's request against the now enlarged access control policy.
6. An administrator may dynamically delegate his administrative role to another person, providing the delegate falls within the scope of the administrative role credential validation policy set by the resource SoA (see Figure 7.1).

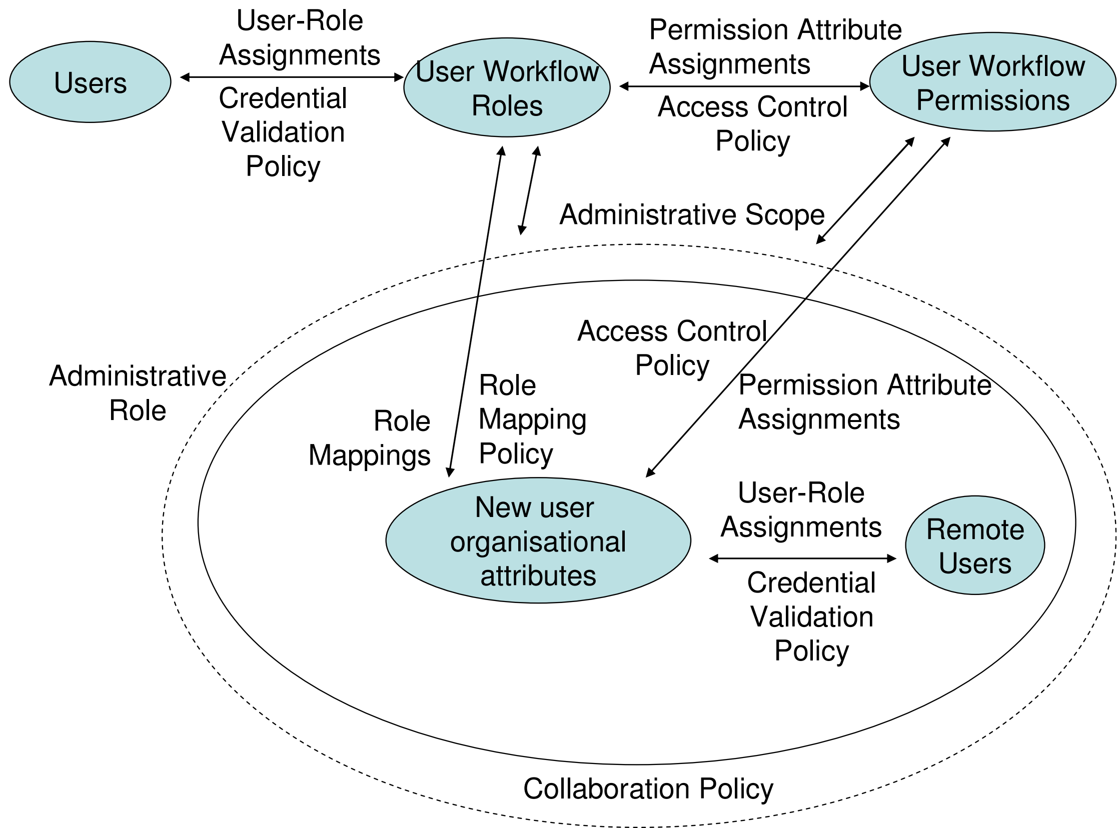


Figure 7.2. Collaboration Policies

7.1 Updating the Collaboration Authorisation Policy

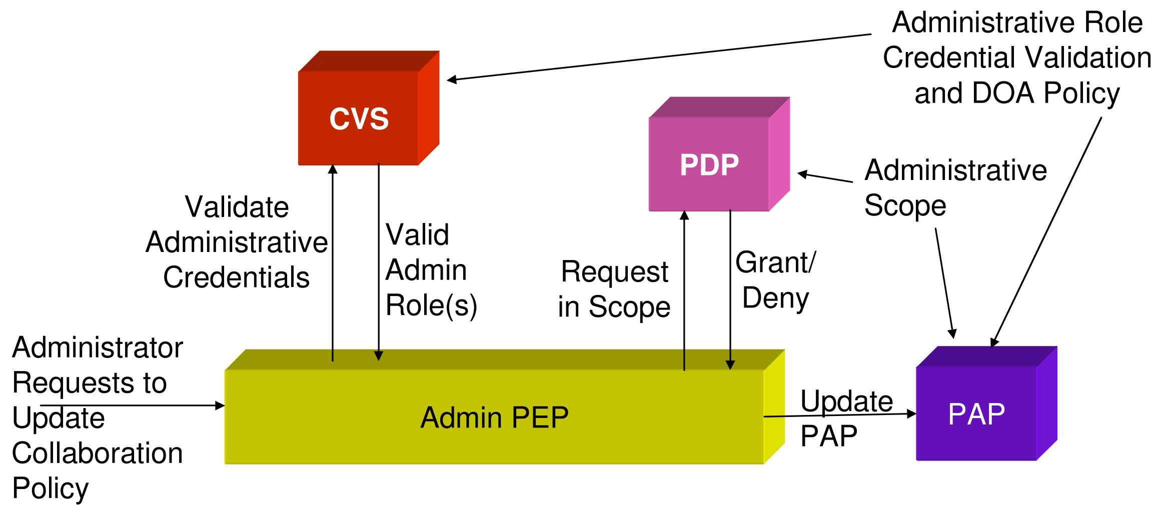


Figure 7.3. Updating the Collaboration Authorisation Policy

The high level conceptual model for the dynamic updating of collaboration authorization policies is presented in Figure 7.3. The PAP holds the authorisation policy of the target domain. The system is initialized by the SoA writing an administrative role validation policy for the CVS and an administrative scope validation policy for the PDP. These policies are stored in the PAP. The CVS and PDP need to read these policies in order to validate the administrators' requests.

The SoA delegates the administrative roles to remote administrators and they can delegate further if necessary. The delegation web service described in section 6 can be used for this, or alternatively, if the administrators have their own key pairs, they may issue their own delegation attribute certificates directly to other administrators. These new administrators are now able to define their own collaboration policies within their administrative scope.

An administrator sends a request to add or update a collaboration policy to the Admin PEP. The request may contain the administrator’s delegated role(s) as signed credentials. To make this policy take effect in the target domain, the Admin PEP requests the CVS module to validate the administrator's claimed administrative role(s) and to return his valid ones. The CVS is either pushed the administrator's administrative credentials, or alternatively it may pull them from a repository. It validates them based on its administrative role credential validation policy and returns the valid administrative roles to the Admin PEP. The Admin PEP now needs to know if the presented collaboration policy is within the administrative scope of the validated roles of the administrator. In order to do this, the Admin PEP creates a request to the PDP to validate either the role-permission assignments or the attribute-role mappings (or both) within the collaboration policy. The subject, action and target of the request are: the set of valid administrative roles, the Map or Assign action, and the local workflow role(s) or permission(s) respectively. If the request is granted, the access control and/or role mapping and credential validation policies will be stored in the PAP. The Admin PEP now informs the CVS and PDP about the new policies that have been added to the system and the CVS and PDP read them in. Any implementation of the Admin PEP, CVS and PDP should be able to perform their tasks automatically without human intervention.

The collaboration policies for one collaboration should be independent from those of all other collaborations, regardless of who is responsible for administering the policies. The consequences of this when evaluating user access requests are that either there should be a separate authorisation system (PDP and CVS) and associated policies for each collaboration, or if one authorisation system (PDP and CVS) makes access control decisions for multiple collaborations, then the policies for each collaboration must be kept separate and not combined. One way of doing this would be to have a unique collaboration ID for each collaboration, and to identify which collaboration each policy applies to and to which collaboration each user access request refers.

The SoA or the administrators who have permission to define a collaboration policy can also revoke an existing collaboration policy. In order to do this, they send a collaboration policy to the PEP along with a revoke request. The Admin PEP queries the CVS to get the valid administrative roles (and thus administrative scope) of a requestor and then queries the PDP in order to confirm that the requestor can define this policy. If he can then the Admin PEP removes the various policies from the PAP and informs the CVS and PDP of the removal. When an administrator's administrative role is revoked, we do not propose to automatically revoke any collaboration agreements that he might have established, since this may not be appropriate. Instead the SoA or replacement administrator can always revoke a collaboration policy by the method just described.

8 Infrastructure for the Distributed Enforcement of Sticky Policies

When users provide their PII to an organization, they need to provide their consent for how the organization can use their PII. These are the users' policies for the use of their PII. If their PII is transferred to a third party, their consent policies should also be transferred to the third party. Organizations also have their own policies for the use of their data, and similarly if they transfer their data to a third party, they also want their policies to be enforced by the third party. Consequently we introduce the concept of sticky policies[8] [19] into policy based authorisation infrastructures. A sticky policy is a policy that is "stuck" or bound to the data to which it pertains, and it should always accompany the data as the data traverses the network. In this way each system that grants access to the data can use the sticky policy that is bound to the data to determine if access should be granted to the data or not.

When a subject requests permission for some data to be transferred to another system, the authorisation decision that is returned from a PDP may contain an obligation along with the grant or deny result. This obligation refers to the sticky policy that is intended to accompany the retrieved data. Obligations are used to return or refer to the sticky policy, as obligations must be enforced by the PEP prior to granting the subject access. However these sticky policy obligations typically wont be fully enforceable by the local PEP. They will need to be transported to the security system (PEP/PDP) of one or more remote sites for processing and enforcement along with the transferred data to which the sticky policy applies. For example, an outgoing message containing personal identifying information (PII) may be allowed to leave the current system, providing that the user's privacy policy is attached to it and that this policy is enforced by the PEPs/PDPs of every receiving system; or an outgoing confidential message may be allowed to leave the current domain providing that its contents are deleted by the receiving system within 7 days of receipt. We thus have the situation where the outgoing message needs to be supplemented with security policy information that is to be enforced by the receiving system. How is this to be achieved?

In this section we present three different possible approaches to solving this problem, which we call the *encapsulating security layer (ESL)* model, the *application protocol enhancement (APE)* model, and the *back channel* model. All three models require the introduction of a new component, the application independent PEP (AIPEP) to be an interface between the conventional application dependent PEP and the existing application independent PDP. The AIPEP acts like a PDP to the

[8] These are policies that should be firmly attached to data, should travel with the data messages throughout a distributed system and should be enforced by each data processing node in the system.

PEP and a PEP to the PDP. The functionality of the AIPEP is to process and enforce the sticky policies and associated obligations that apply to multiple nodes of a distributed application. In addition, in the ESL model it transports the application messages (see Figure 8.1), whilst in the back channel model it transports the policies (see Figure 8.4). The functionality of the PDP remains the same as in today's systems in all models (and hence the PDP remains application independent) whilst the functionality of the PEP may have to be modified to fit the new requirements of the AIPEP.

8.1 The Application Protocol Enhancement (APE) Model

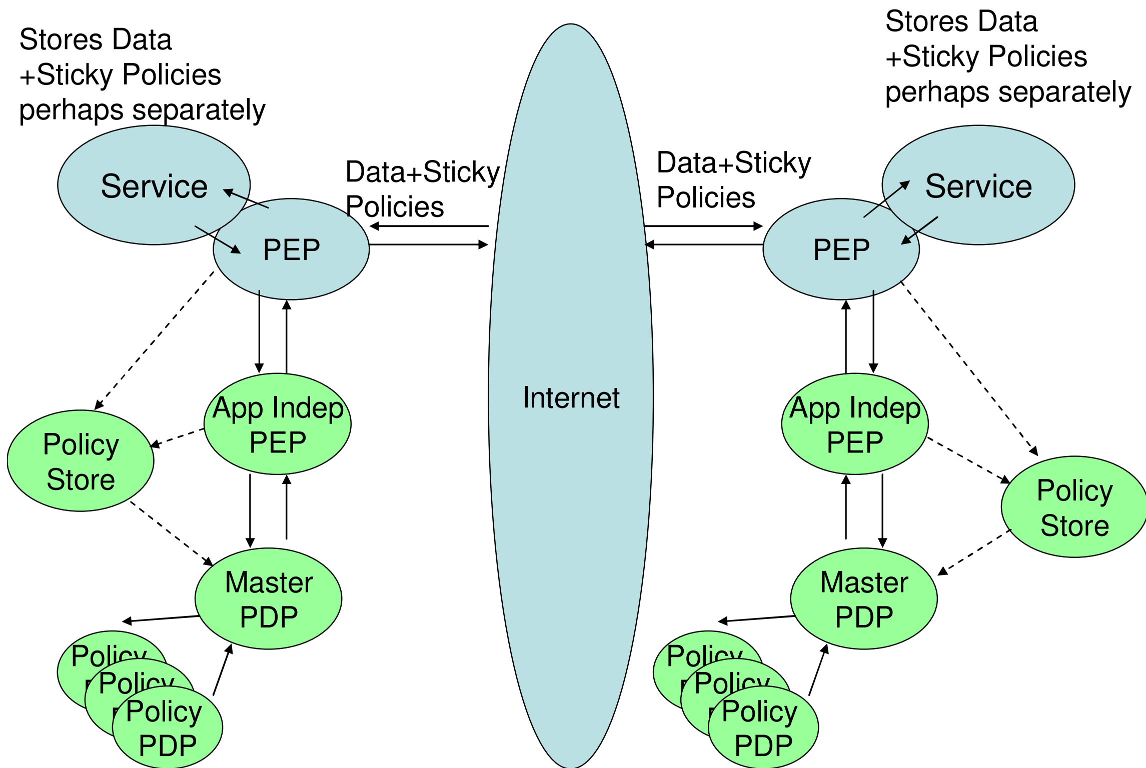


Figure 8.1. The Application Protocol Enhancement Model

In the APE model, the PEP supplements the existing application protocol with policy information. In a situation such as callout 1 or 3 in figure 2.2, the PEP receives and parses the outgoing message that is to be security enforced, segments and extracts relevant information from it (see Section 8.4), and passes this to the AIPEP for an authorisation decision. The AIPEP passes the

request to the Master PDP. The Master PDP calls the relevant subordinate PDPs and returns the overall decision to the AIPEP. If this is Deny, this is relayed to the PEP whereupon this message segment should be discarded from the outgoing message and not transferred[9]. If this is Grant, the Master PDP optionally returns an obligation to the AIPEP saying what policy should accompany the outgoing message segment. The AIPEP passes this obligation to the PEP along with the authorisation decision, and the latter produces a policy packet that has to be attached to the outgoing message segment in an application dependent manner. This could be inline using an existing application protocol provided container or an existing extension point in the application protocol. Another approach could be to annotate all relevant application protocol data elements with an xml:id attribute (this is often possible in many application schemas) and then supply in a SOAP header (e.g. <tas3:StickyPolicies>) the sticky policies, referencing the xml:id attributes of the data. The SOAP header approach is similar to the back channel approach, see later, but instead of an explicit back channel, uses the SOAP headers as a back channel. The actual contents of the sticky policy packet are transparent to the PEP, but should be internationally standardised so that all AIPEPs and PDPs can understand it. We propose a schema for this in section 5 Figure 5.2. The PEP duly attaches this policy packet to the outgoing message segment, and may merge several segments together again before sending the message to the recipient system.

[9] A use case for this is where a set of patient records containing details of recent surgical operations are being transferred to a researcher for analysis, but the records include those of VIPs, which should be removed from the outgoing results.

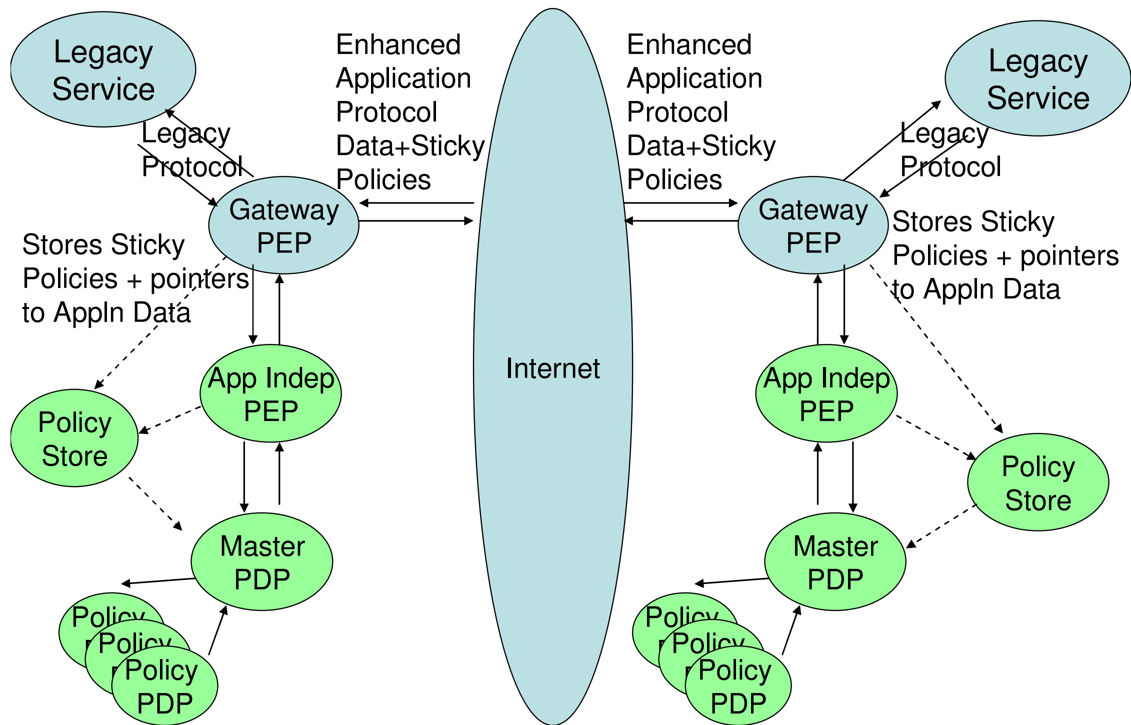


Figure 8.2. Using a Gateway for Legacy Applications

Some applications, such as S/MIME [20], are able to attach policies to data as they already have protocol fields that can be used for this. Other applications may not have such flexibility. In this case the PEPs could run in gateway machines and the communication between the application service and a PEP could be the existing legacy protocol, and the protocol between the PEP gateways could be an enhanced application protocol (see Figure 8.2). The Gateway approach effectively corresponds to the mediation box or filter model described in section 4 "Deployment and Integration Models" of D2.4 [46].

When the message is received by the PEP at the receiving system, the PEP parses the incoming message, extracts the policy packets, message segments and relevant information for the PDP (see Section 8.4) and passes these to the AIPEP. The AIPEP parses this policy and processes it. Whilst the incoming policy processing is fully carried out by the AIPEP on message receipt, it may have knock on effects which will cause a similar policy packet to be attached to the same information when the latter is subsequently transferred elsewhere, or it may even deny the information from being transferred to other specific destinations. For example, if the policy packet

is a user's privacy policy containing some blacklisted sites and the information is the user's PII, the AIPEP may update the PDP's policy to include a policy rule denying access to the blacklisted sites, and an obligation that says when this PII is granted permission to leave the local system the same sticky privacy policy should be attached to the data before it leaves the system.

When the AIPEP has finished processing the incoming policy, it calls the Master PDP for an authorisation decision on the incoming message segment. The Master PDP calls the subordinate PDPs, then makes the overall decision and if granted (or denied) may optionally return a set of obligations to the AIPEP. These obligations can contain any set of actions which are to be enforced locally by the AIPEP or PEP when it is processing the incoming message (as now). The encodings of the various obligations are such that the AIPEP knows which obligations it can process and enforce, and which it should return to the PEP for it to process as described in section 2.5. For example, in XACML [15], each type of obligation is given a unique URI. Note that if the Master PDP denies permission for the incoming message segment to be received, then the AIPEP will need to rollback the effects of any policy actions it has taken (such as updating the PDP's policy).

8.2 The Encapsulating Security Layer (ESL) Model

In the ESL model, the security layer is a protocol layer beneath the application layer. Each application layer protocol message segment is passed to the AIPEP, which wraps each message with its own security header and policy and then transfers the message to the AIPEP at the remote system, which strips off the security header, enforces the policy, and passes the message segment back to the application layer. This is similar to the SSL approach, only in this case the security layer is responsible for carrying policies between systems rather than MACs and encrypted messages.

In more detail the system works as follows. The PEP parses the outgoing message that is to be enforced, segments and extracts relevant information from it (see Section 8.4), and passes the message segment and the extracted information to the AIPEP for handling. The PEP must also pass the connection details of the recipient application system (i.e. the remote application endpoint) with the first message segment. The AIPEP calls the Master PDP - note that this is the same Master PDP as in the APE model - and receives an authorisation decision, which if denied, causes a deny to be returned to the PEP. The PEP can then decide whether to send the message minus the offending segment, or terminate the entire message. If granted, the Master PDP may optionally return an obligation, which is processed by the AIPEP to create a policy packet for attachment to the application message segment.

In the ESL model, the AIPEP can understand the standardised policy contents and therefore can potentially treat different outgoing messages in different ways e.g. encrypt some, sign some,

use SSL etc, which is not something that the PEP in the APE model could do. The AIPEP sends the outgoing message to its peer AIPEP, and includes details of the recipient application system (i.e. application endpoint) with the first message. The AIPEP at the receiving system strips off the security header and policy, enforces the policy (as before), then calls the Master PDP (as before). The application message is finally passed to the PEP at the specified application endpoint.

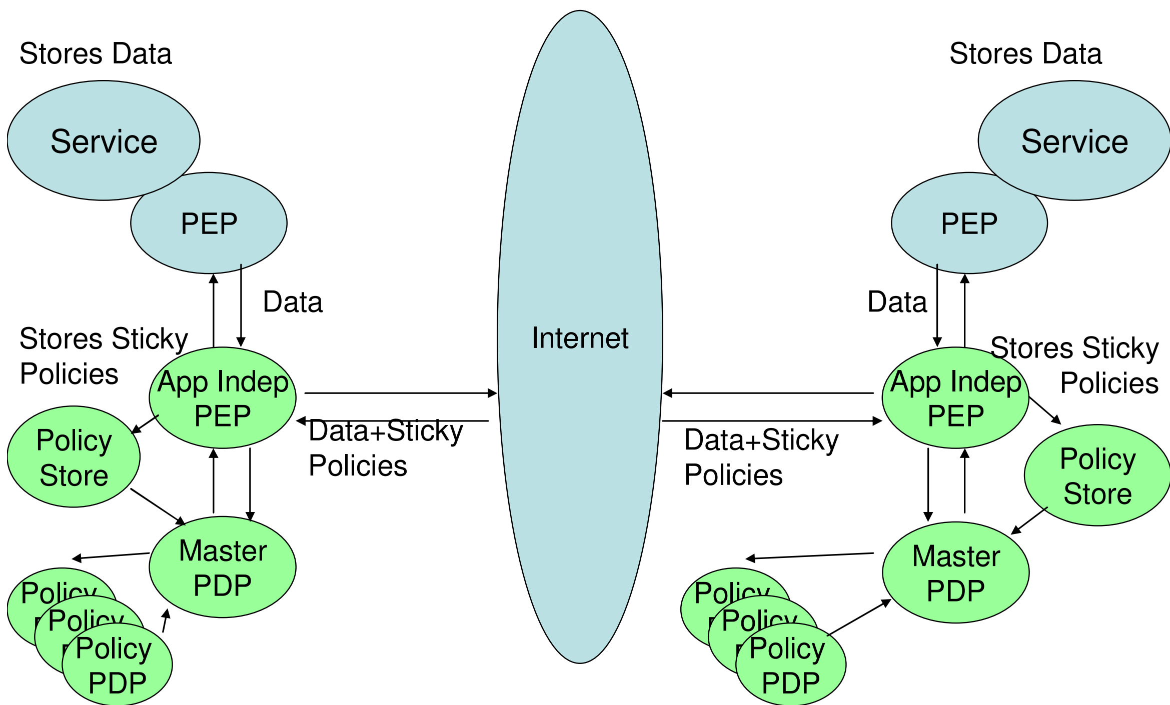


Figure 8.3. The Encapsulating Security Layer Model

Note that the AIPEPs will need to have their own standardised protocol and ports, so that they can talk to each other across the Internet. There will also need to be a mechanism for computing the endpoint information of an AIPEP given the application dependent endpoint information, as the latter is passed by the PEP on the sending side to its AIPEP. One solution to this discovery problem is described in section 2.5 "Realization of the Discovery Function" in D2.4.

8.3 The Back Channel Model

This model is least perturbing to existing distributed applications, since the AIPEP establishes a back channel with its peer AIPEPs in order to transfer the obligations and sticky policies that must accompany the application data.

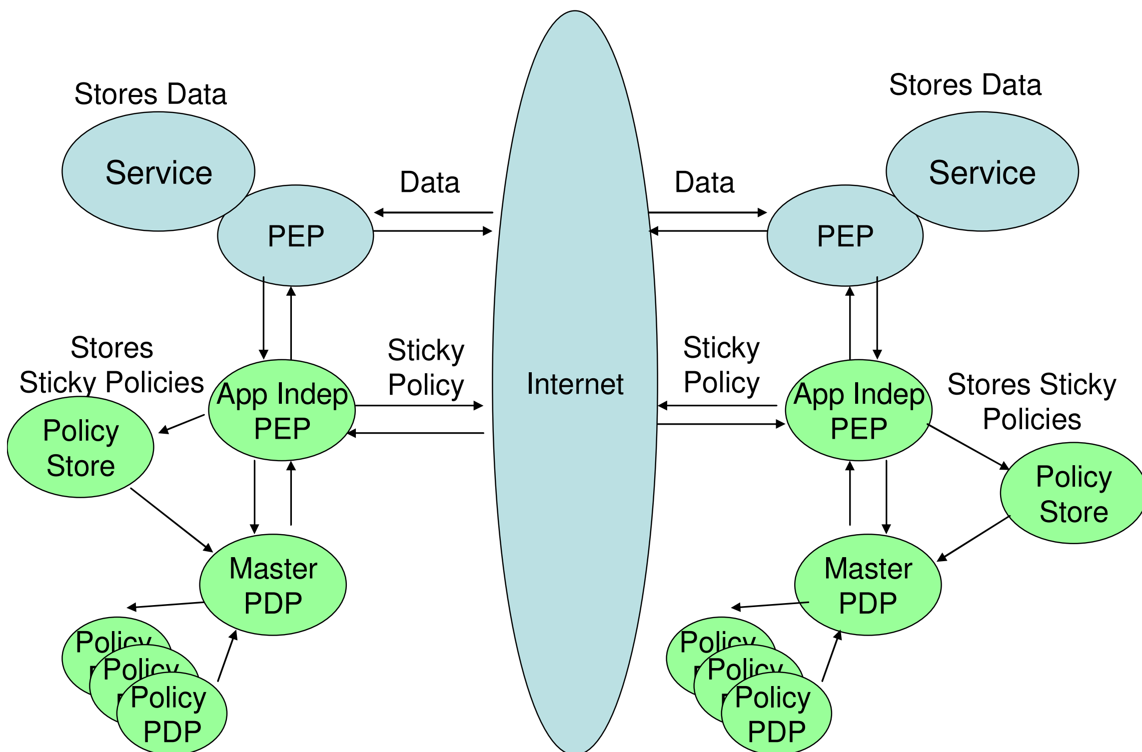


Figure 8.4. The Back Channel Model

The application works exactly as it does today, making a callout to the application independent infrastructure for a decision and enforcing the decision along with any local obligations that are returned. The application transfers outgoing application data to its peer without needing to modify the application protocol or data stream (except to remove any data blocks that are denied outgoing access). The AIPEP relays the decision request to the Master PDP and if an obligation to attach a sticky policy to the outgoing message is returned, the AIPEP needs to transfer this policy to its peer AIPEP before returning the grant decision to the PEP. There are two main issues here. Firstly, how does the sending AIPEP know who the receiving AIPEP should be? Secondly, how does the receiving AIPEP know which data block this policy applies to? The answer to the first question is that after trust negotiation has successfully completed (see Section 8.6) the trust negotiation module notifies the AIPEP with details about the remote trusted party. The answer to the second question is to attach filtered components of the

request context received from the PEP to the sticky policy, since the request context uniquely identifies the data block about which the decision has just been made (see section 8.4).

When the receiving PEP receives the incoming message and calls the AIPEP for a decision by passing it a request context, the applicable sticky policy should be ready and waiting for it, so that processing can continue as in the APE model. Each AIPEP will need a temporary cache to store incoming sticky policies so that they can be linked to their data segments/request contexts when these subsequently arrive. There will be timing issues to consider, say if the application data arrives before the sticky policy because the latter uses SSL whereas the application does not. These will need to be addressed during the implementation phase.

8.4 Functionality of the PEP

The standard information [2, 15] that the PDP (and hence the AIPEP) needs to be passed by the PEP is: details about the requesting user (i.e. subject attributes), details about the application destination (i.e. resource attributes), details about the requested action (i.e. action attributes) and any other relevant information (i.e. environmental attributes). In XACML, this is called a **request context**. Each application will typically store, format and transport this information in an application specific way, but the AIPEP needs it to be passed in a standard application independent way (for relaying to the PDP). The PEP is the only security entity that is capable of parsing an outgoing application message since it is the only security entity that understands the contents of the application specific message.

So a primary function of the PEP is to parse each application message that needs an authorization decision, extract from it the information that the PDP requires (i.e. the request context), and format this into the standard format required by the AIPEP/PDP. In the TAS3 project we will use the standard request context format specified in XACML. The precise information requirements of the PDP are dependent upon the application specific policy that is being used to make authorisation decisions. Consequently this information has to be determined in an application specific way since no two applications will require the same sorts of policies e.g. a mobile application might require authorisation decisions to be made based on the locations of the various mobile devices, whereas a printing application may have no such requirements but may instead require the time of day to be taken into consideration. Consequently, the precise contents of the information which the PEP has to extract from its messages to pass to the AIPEP/PDP have to be agreed by each application developer when the application is being built.

All the above models require the PEP to be responsible for parsing and logically segmenting outgoing messages into appropriate security blocks (and creating matching request contexts) so that each block can have a sticky policy applied to it. Whether the PEP passes the sticky policy to the AIPEP with the request context, or the AIPEP picks up the policy from its local storage, nevertheless it is always the responsibility of the PEP to provide the link between the policy and

the security block/request context. A security block is an atomic unit of application data from a security perspective, which has a sticky policy attached to it (either physically or logically depending upon the model). A security block is defined as any application message, containing any arbitrary number of application elements, which has a common security requirement.

Consequently, each security block may have a different security policy stuck to it. It is an application dependent matter to define what constitutes a security block. For example, the application may be transferring the names and addresses of a group of people in a single application message. In one application, each person may have the ability to set their own privacy policy for their own PII. At the application level, the group of names and addresses may be considered a single application level message for transfer to a remote site, but at the security level, each name and address may be considered to be a separate security block and therefore needs its own sticky policy. Thus it is the responsibility of the PEP to parse the outgoing message and to separate it into multiple security blocks, and to call the AIPEP once for each block (i.e. name and address tuple in the message). In this way different privacy policies can be stuck to different name and address tuples.

In another application a single corporate privacy policy may control access to the PII (names and addresses) of everyone in the database. In this case the entire message would be treated as a single security block and one privacy policy would be attached to the whole group of names and addresses. An application may transmit a large message with multiple elements as one security block, but the sticky policy might only refer to one particular element in the message. This is achieved by parsing the message, creating the request context from the one particular element, and then sticking the policy to the entire message. For example, a complete ePortfolio may be transmitted as a zip file accompanied with a sticky policy saying "The data from file A_identification_Pete.xml at XPATH location learnerinformation/identification/address must be used only for APEL purposes." Consequently it is the responsibility of the PEP to decide what constitutes a security block from the application's perspective and to act accordingly when calling the AIPEP. The result is that the AIPEP always receives the request context and either the sticky policy that applies to it or a reference to the policy.

8.5 Trust Negotiation

One issue that needs to be addressed by all three models is how does the sending system know if the receiving system can be trusted to obey the sticky policy that it receives? We propose that the well researched topic of trust negotiation [21] is used for this. Trust negotiation relies on trusted Attribute Authorities to issue credentials to components of a distributed application, in the form of signed attribute certificates or assertions, and during trust negotiation both the sender and recipient determine if the other party has the necessary credentials to participate in the interaction. We propose that trust negotiation is carried out by an application independent

module during the process of service provider selection, prior to the transfer of the first application layer protocol message. In this respect, trust negotiation is independent of any of the models described here and of the application layer protocol, and is a necessary precursor of any application message and sticky policy transfer. Once trust negotiation has successfully completed, the trust negotiation module is in possession of all the necessary details about the remote party in order for the AIPEP or PEP to connect to its peer entity.

Note that in the ESL model, it would be possible to build the trust negotiation functionality into the AIPEP rather than having it as a separate component. The peer AIPEPs would negotiate with each other to determine whether each is trusted or not, before the application data is transferred. If trust cannot be established by the AIPEP, the PEP would be informed that the application data cannot be transferred to the chosen application endpoint. The PEP would then be responsible for selecting another service provider/application endpoint and asking the AIPEP to try again. In the APE and back channel models, the PEP will need to call the trust negotiation module during service selection, prior to calling the AIPEP.

A full description of the trust negotiation protocol and implementation is given in sections 11 and 12.

9 Event handling infrastructure & its application to adaptive audit controls

The TAS[3] infrastructure will incorporate an event handling infrastructure based on the publish/subscribe paradigm. Publish/subscribe is an asynchronous messaging passing infrastructure in which messages are grouped into classes. Unlike conventional messaging systems, in which senders send messages to specific receivers, in publish/subscribe messaging, senders (also known as publishers) send particular classes of message to receivers (also known as subscribers) who are interested in them. Consequently a message sender does not know how many recipients there might be for its messages; it all depends upon how many recipients have currently subscribed to receive that particular class of message. Subscribers express interest in one or more classes of message, and only receive messages that are of interest to them, without knowledge of what (if any) publishers there currently are.

Publish/subscribe messaging will be used by the TAS[3] infrastructure to send messages about specific security, privacy and trust related events. For example, if an administrator updates an authorization policy, a PDP can be sent an event message to inform it of the fact, so that it can read in the latest policy. The PDP will be a subscriber for messages of this class, and the policy management software will be a publisher of these messages. Publish/subscribe allows us to decouple system components, so that the components will work correctly regardless of whether there is another component sending or receiving messages or not.

A primary area in which we intend to use publish/subscribe is adaptive audit controls. Each TAS[3] system component that sends log information to the secure audit web service (SAWS) [22] will subscribe for particular security/privacy/trust events, such as a security alert, or an occurrence of break the glass, and this will cause it to increase its level of logging that it sends to SAWS. Conversely, when an “all clear” or “glass re-set” message is transmitted, the component will reduce its level of logging to the minimum. Log4J is a Java package that already supports multiple levels of logging. By incorporating SAWS into Log4J as a repository to accept Log4J log messages, we can enable applications to dynamically alter the amount of logging they send to SAWS, by them simply switching between the different Log4J logging levels.

Researching into publish/subscribe mechanisms is not an objective of the TAS[3] project, and we do not propose to either design or spend significant resources developing our own publish/subscribe message passing infrastructure. We propose to use any suitable existing open source publish/subscribe messaging product that is available. (See D2.4 [46] section 2.7.1 "Audit Event Bus" for descriptions of the current choice.) We expect to have to enhance existing software by adding security features to suit our requirements. A full description of the secure event handling and publish/subscribe infrastructure will be given in D8.2 [32].

10 Authorization Ontology

10.1 Overview

This document describes the design of an ontology model for authorization policies. The ontology model is related to the common ontology suggested in WP2 (in deliverables D2.2 [48] and D2.3 [49]) but it is not yet fully aligned with these, since they are still evolving. We propose to fully align the authorisation policy ontology with the common ontology in the next version of this deliverable.

Figure 10.1 shows the top layer of the common ontology. The main assumption is that the world (or T) contains three main concepts, namely Entity, Predicate (previously called Activity), and Descriptor. An Entity represents anything that can take part in an action or that can be acted upon. An Activity (or Predicate) denotes a verb which affirms or denies information about an Entity, while a Descriptor categorises or describes either an Entity or a Predicate..

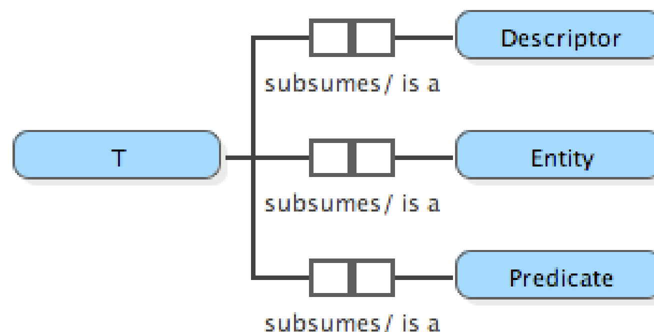


Figure 10.1. The top layer of the Descriptive Upper Ontology

The authorisation ontology model comprised three parts. The first part is the information representing the system to be governed by the authorisation policy; the second part is the policy rules that are used to get an authorization decision result for an authorisation decision request to a specific system; and the third part is the authorisation decision request. All these classes are defined under three top level classes, and the full class hierarchy can be found in the ontology model section.

The major classes used to describe the system include Subject, Action, Resource, Environment and Attribute. The Action class represents operations (with parameters) that can be performed on resource in a system. Resources are those valuable assets to be protected within a system.

Subjects are those entities who perform Actions on resources. Environment is additional information about the system e.g. the current time, current state variables etc. Descriptors and Attributes are used to describe features of Subjects, Actions, Resources and Environment. An authorization decision result can be made based on the descriptors and attributes attached to Subjects, Actions, Resources and the Environment.

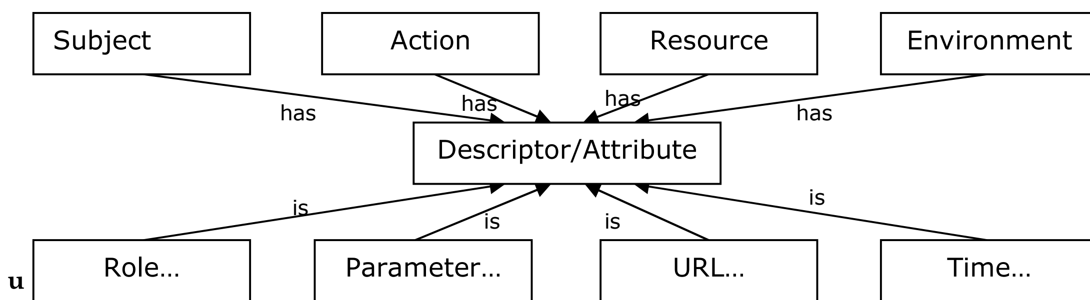


Figure 10.2. The System

The policy consists of a set of rules. Each rule is structured in the form of "if (Predicate) then (Result)", where Predicate is a statement that can be evaluated as either TRUE or FALSE by a PDP, and Result can be a Decision with optional Obligations.

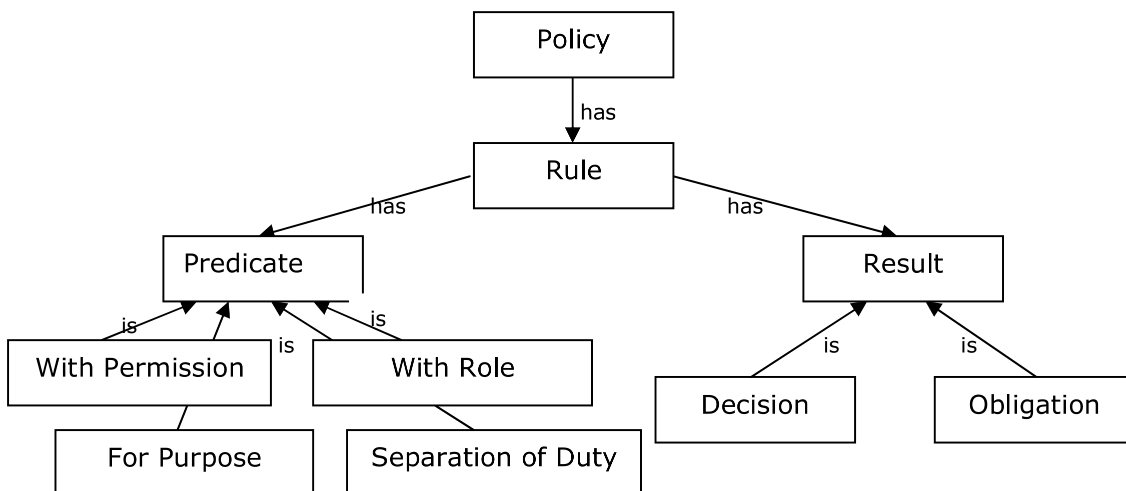


Figure 10.3. An Authorisation Policy

Predicates are assertions about the attributes of a system. They can be assertions about or constraints upon the subject attributes, action attributes, resource attributes or environment attributes. They can also be statements of privacy, or constraints based on previous actions such as separation of duties (SoD). New predicates can be defined providing they can be evaluated by the PDP. A Predicate can include other Predicates as sub-predicates; a binary property of a Predicate indicates whether All sub-predicates or Any sub-predicate should be satisfied. A rule may include more than one predicate; only when all predicates are satisfied is the Result part of the rule returned by the PDP. There are two components of a result: Decision and Obligation(s). For an authorization result, there is one and only one Decision result, and any number of Obligations are optional.

A policy may contain more than one rule. Each rule has a priority, which is an integer ranging from 0 to 2^16; the higher the number, the higher the priority. There are two built-in rules with lowest priority 0, PermitAll and DenyAll. They are mutually exclusive and can be used to construct by-default-permit or by-default-deny policies.

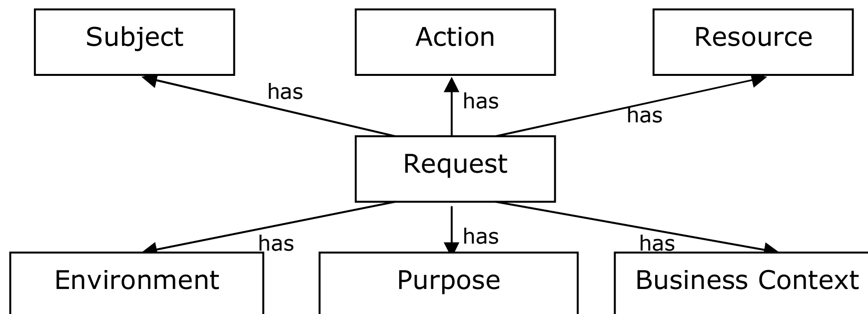


Figure 10.4. An Access Request

An access Request is modelled as a class in the ontology model. An access request may include Subject, Action, Resource, Environment, Purpose and Business Context. These objects and the descriptors/attributes associated with them will be used by the PDP to evaluate the predicates in the policy.

10.2 The Authorisation Ontology Model

In this section of the document, the authorisation ontology model will be described in an Object-Oriented style, rather than a Description-Logic style. This means that the model is structured in terms of classes, properties and instances. The hierarchy of all classes is shown in Figure 10.5. The description of each class is given below.

AccessRequest
AccessRequest is a Request by a Subject, who requests to perform an Action on a Resource for a Purpose. Other information that may be included in an AccessRequest can be the information of the Environment and the Context of the Action.
Properties:
Access_has_BusinessContext: Under what Context the Action is performed.
Access_has_Environment: The information of the Environment when Request is submitted
Access_has_Purpose: For what Purpose the Subject requests to access that Resource.

AccountObligation
AccountObligation specifies under which account the Action should be performed.
Properties:
underAccount: Under which account the Action should be performed. This could be a Subject or an Identity.

Action
Actions are operations allowed to be performed in a system.
Properties:
Action_has_Parameter: An action may have zero or more Parameters.
Action_on_Resource: This indicates on which resources an action can be performed on.

Activity
Imported from common ontologies defined by WP2.

Administration
Administration is a type of function like "do ACTION with ATTRIBUTE on USER/RESOURCE". An example is "ASSIGN the STAFF ROLE to ALICE". It can be associated with an Administrator, to state what privileges an Administrator has.
Properties:
onUser: The User that the AdministrationAction to be performed on.
performAdminAction: Defines which AdministrationAction to be performed in an Administration function.

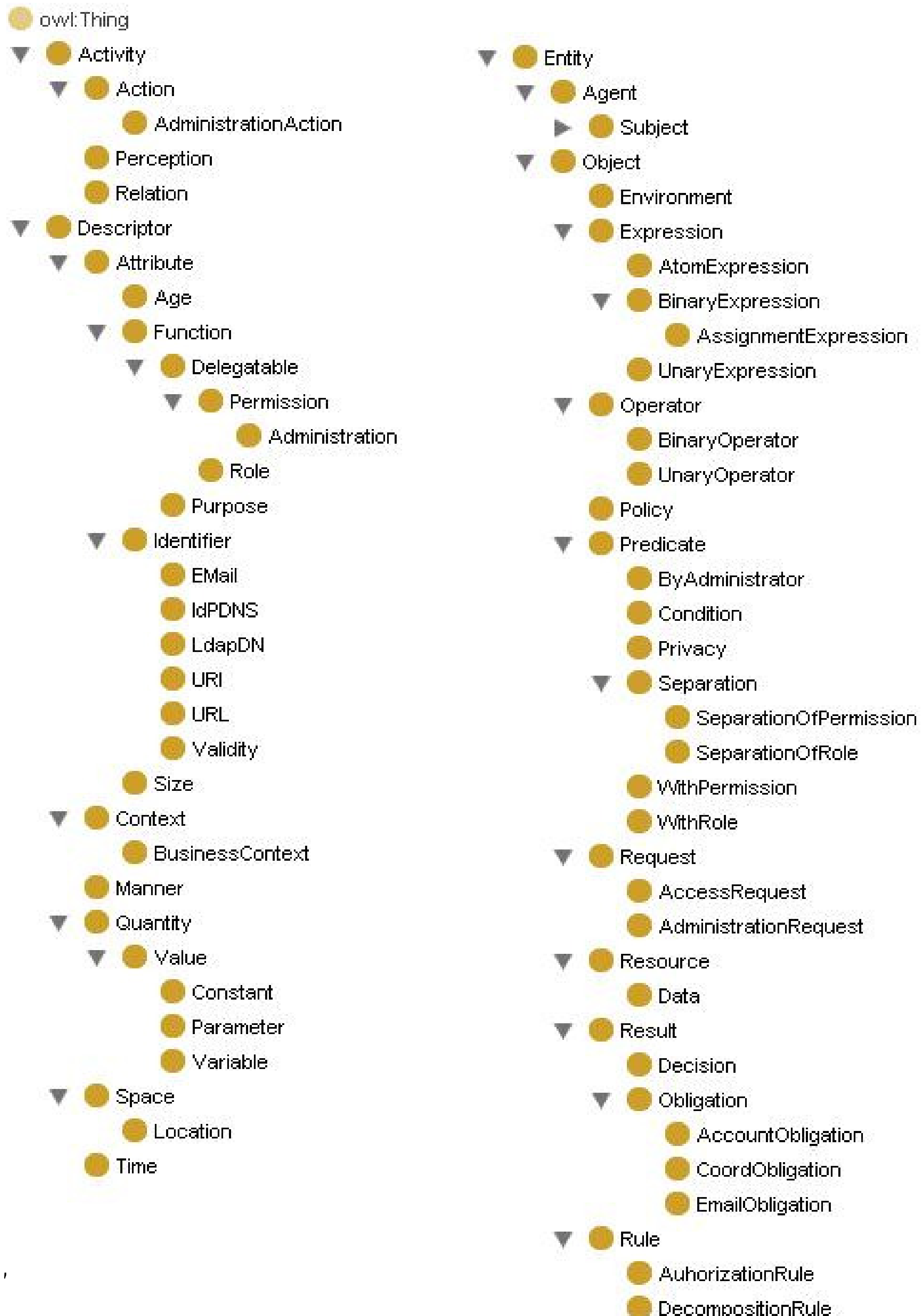

owl:Thing
Activity
Action
AdministrationAction
Perception
Relation
Descriptor
Attribute
Age
Function
Delegatable
Permission
Administration
Role
Purpose
Identifier
EMail
IdPDNS
LdapDN
URI
URL
Validity
Size
Context
BusinessContext
Manner
Quantity
Value
Constant
Parameter
Variable
Space
Location
Time
Entity
Agent
Subject
Object
Environment
Expression
AtomExpression
BinaryExpression
AssignmentExpression
UnaryExpression
Operator
BinaryOperator
UnaryOperator
Policy
Predicate
ByAdministrator
Condition
Privacy
Separation
SeparationOfPermission
SeparationOfRole
WithPermission
WithRole
Request
AccessRequest
AdministrationRequest
Resource
Data
Result
Decision
Obligation
AccountObligation
CoordObligation
EmailObligation
Rule
AuhorizationRule
DecompositionRule

Figure 10.5. The Authorisation Ontology Class Hierarchy

AdministrationAction
AdministrationAction is a type of Action. It can be performed on Users or Resources, with Attributes as parameters.
Instances:
Delegate, Issue, Revoke

AdministrationRequest
AdministrationRequest is a Request to perform AdministrationActions on Users or Resources, such as issue or revoke an Attribute.

Administrator
Administrators are a kind of Subjects who can perform AdministrationActions in a system.
Properties:
Administrator_do_Administration: This defines Administration tasks associated with an Administrator.

Agent
Imported from common ontologies defined by WP2.

AssignmentExpression
AssignmentExpression is a type of Expression, which assign the value of right-hand side element to left-hand side element.

AtomExpression
AtomExpression is just an Attribute or a Value.

Attribute
Attributes are used to describe Entities and Activities in a system. They can be used to evaluate Predicates in rules.
Properties:
attributeValidity: An attribute may have a Validity period. Only within the period, the Attribute is valid.
issuedBy: This shows by which Subject this Attribute has been issued.

AuthorizationRule
Rules for Authorization. This kind of rules has one and only one Decision result, with optional Obligation result.
Properties:
Auhtorization_has_Decision: The Decision result of an Authorization rule.

Authroization_has_Obligation: Optional Obligations with the Decision result.
Instances:
PermitAll, DenyAll

BinaryExpression
A BinaryExpression has a BinaryOperator and two elements.

BinaryOperator
BinaryOperator is a type of Operator can take both LHS and RHS, such as Or.
Instances:
Assign, Divide, Sub, And, Equal, Or, Multiply, Add

BusinessContext
BusinessContext defines the boundary or a set of Actions performed in the system. Constrains like SoD (Separation of Duty) must be defined within a Context.
Properties:
contextString: A string used to identify a Context.
firstAction: The start Action of a business context.
lastAction: The end Action of a business context.

ByAdministrator
This Predicate states that a Request must be submitted by a certain Administrator.
Properties:
Assertion_on_Administrator: By which Administrator the administration tasks are taken by.

Condition
Condition is represented by boolean Expression, which tells when this Condition is TRUE or FALSE.
Properties:
conditionExpression: A boolean Expression of the Condition.

Constant
Constant is a value will not change.

Context
Imported from common ontologies defined by WP2.

CoordObligation
CoordObligation enables assigning an Expression to a Variable. For example, balance = balance - amount. This is represented by an AssignmentExpression.

Data
Data is a type of Resource, which can be further used by the requester. To protect Data from being misused, Privacy objects can be associated with Data to specify this Data can only be used for certain Purposes.
Properties:
dataPrivacy: Privacy constrains associated with a Data.

Decision
Decision is the Result of an Authorization rule; it could be Permit, Deny, NotApplicable or other type of Decision depending on the PDP implementation.
Instances:
Permit, Deny, NotApplicable

DecompositionRule
Decomposition rules are used to refine policies in high level to concrete policies which can be used for decision making.

Delegatable
This is used to represent a function which can be delegated to others. It only can be hold by a subject. When an attribute is delegated, its delegationDepth is deducted by one.
Properties:
delegationDepth: The depth an attribute can be delegated.

Descriptor
Imported from common ontologies defined by WP2.

EMail
An Email address.

EmailObligation
EmailObligation specifies an Email should be sent out in conjunction with the enforcement of a Decision.
Properties:
emailBody: The body of the Email to be sent out.
emailFrom: This specifies through which account the Email should be sent out.
emailSubject: The subject of the Email.
emailTo: To whom the Email should be sent.

Entity
Imported from common ontologies defined by WP2.

Environment
Environment includes the information, which is not part of the Request but is needed for making Decision. For example, current time of the system.
Properties:
Environment_has_Attribute: Attributes used to descripe an E nvironment.
Environment_has_Location: The Location information of an Environment.
Environment_has_Time: Current time of an environment.

Expression
General form of Expression used in a policy. Each Expression may have one operator, and one or two elements depending on the type of Operator. An expression can have just one element of an Attribute or a Value, in this case it is an AtomExpression. An Expression can also be nested as an element in other Expressions.
Properties:
expressionType: The value type of the Expression.
lhs: The first or left-hand side element of an Expression.
operator: The operator of an Expression.
rhs: The second or right-hand side element of an Expression.

Function
Functions represent combinations of Actions and Resources. Depending on entities a function is used to descript, it could be a permission, a role or a purpose. A function can include other functions, to form a function hierarchy.
Properties:
Function_comprises_Function: A function may comprise other functions to form a function Hierarchy.
onTarget: The target of the Action in the definition of a Function.
performAction: The Action to be performed in the definition of a Function.

Identifier
Identifier is used to identify a Subject or a Resource.
Properties:
identifierString: The string value of the identifier.

IdPDNS
The DNS name of a Shibboleth Identity Provider (IdP) server.

LdapDN
A DN (Distinguished Name) in a LDAP directory.

Manner

Imported from common ontologies defined by WP2.

Object
Imported from common ontologies defined by WP2.

Obligation
Obligation is an operation that should be performed by the PEP in conjunction with the enforcement of an authorization Decision.
Properties:
attributeAssignment: An AssignmentExpression representing the Variable and the value to be assigned to it.
fulfillOn: This specifies the Obligation should be performed in conjunction with which Decision. For example, an AccountObligation should be fulfilled on a Permit decision.
temporalType: TemporalType specifies at which stage of the enforcement of a Decision, the Obligation should be performed. There are three possible values of temporalType: Before, With or After. For example, the temporalType of an AccountObligation should be with.

Operator
Operator is used to construct expressions.

Parameter
Parameters of Actions can be used as Values in Expressions.
Properties:
parameterName: The name of a Parameter.

Perception
Imported from common ontologies defined by WP2.

Permission
A Permission attribute represents a subject is allowed to perform an Action on a certain Resource.

Policy
A policy consists of a set of Rules. When a Request comes to a PDP, the Predicate parts of these Rules are tested with the information from the Request. The Rules will be checked in the order of priority. If the Predicate part of a Rule is satisfied, then the Result part of that Rule will be returned by the PDP.
Properties:
Policy_has_Rule: Rules included in a Policy.
policyName: The name of a Policy.

Predicate

A Predicate is a statement can be evaluated by a PDP, with result of either TRUE or FALSE. The way of defining and evaluating a statement can be customized/extended according to the implementation of a PDP. A Predicate can be included in other ones as a sub Predicate.
Properties:
Assertion_on_Thing: This is a general form of assertion of a certain thing. It should be specialized in each type of Predicate.
satisfyAllOrAny: This indicates how sub Predicates should be satisfied, it can be either satisfyAll or satisfyAny.
subPredicate: Sub Predicates are those Predicates to be satisfied, besides the statements in the current Predicates.

Privacy
Stands for Privacy constrains on Data resource.
Properties:
forPurpose: For what reason a Data resource can be used.

Purpose
Purpose represents the reason of a Request to Data.

Quantity
Imported from common ontologies defined by WP2.

Relation
Imported from common ontologies defined by WP2.

Request
A Request to be answered by a PDP. A Request contains variant information which can be used by a PDP to evaluate Predicates. The PDP will check the Rules of a Policy with the information contained within the Request, and return the Results of the matching rule.
Properties:
Request_has_Target: On which target the Subject requests to perform Action on.
Request_has_Action: The Action that the Subject requests to perform on the Resource.
Request_has_Subject: The Subject who submits the Request.

Resource
Resources are those entities where Actions can be applied on. A set of Attributes can be associated with Resources. These Attributes can be used to construct Policies protecting the Resources.
Properties:
Resource_has_Attribute: Attributes associated with a Resource.

Resource_has_Identity: The Identity of a Resource.

Result
Result is the consequence of a Rule if the Predicate part of it is TRUE. For an Authorization rule, it must have one Decision as its Result, together with Optional Obligations. For other kind of rules, such as reasoning rules like DecompositionRule, the result could be another Predicate which can be further used in other Rules.

Role
Role is a type of Attribute. In RBAC model, a Role is associated with a set of Permission. A user with a certain Role is allowed to perform Actions on Resources defined by the Permissions associated with that Role. Like some other Attributes, Role can be issued to a User by an Administrator of the system.
Properties:
Role_has_Permission: Permissions associated with a Role.
superiorTo: If Role A is superior to Role B, then Role A has all Permissions associated with Role B. This relationship is transitive.

Rule
Rule is the basic element in a Policy. Each rule has two parts, Predicates and Results. The Predciates can be evaluated by a PDP, and if the result is TRUE, then the Results are returned by PDP.
Properties:
Rule_has_Predicate: The Predicate part of a rule. If there are more than one Predicates, all of them should be satisfied before the Results are returned. If a Rule needs "any predicate, then sub Predicates with "SatisfyAny" switch should be used.
Rule_has_Result: The Results part of a Rule. This part is returned by PDP if all Predicates are satisifed.
rulePriority: The priority of a Rule. It is an integer number ranging from 0 (the lowest priority) to 2^16 (the highest priority).

Separation
This is an abstract class representing Separation of Duties (SoD).
Properties:
inContext: In which Context the SoD constrains apply.
maxCardinality: The maximum cardinality of mutually exclusive objects a user may have in single Context.

SeparationOfPermission
This kind of SoD constrains state that in a certain Context, a User must not have more than maxmum number of Permissions from defined mutually exclusive Permissions.

Properties:
mutuallyExclusivePermission: Mutually exclusive Permissions in a Context.

SeparationOfRole
This kind of SoD constrains state that in a certain Context, a User must not have more than maxmum number of Roles from defined mutually exclusive Roles.
Properties:
mutuallyExclusiveRole: Mutually exclusive Roles in a Context.

Service
Services are parts of a system. They can perform actions with or with our user interaction. In some scenarios, Services can also be used to issue/revoke Attributes to other Subjects.

Space
Imported from common ontologies defined by WP2.

Subject
Subjects are entities who perform actions in a system. A set of Attributes can be associated with Subjects. These Attributes can be used to support evaluating Predicates in a Policy.
Properties:
Subject_has_Attribute: Attributes associated with a Subject.
Subject_has_Function: Subjects are associated with a set of Functions, in forms of Permission, Administration or Role.
Subject_has_Identity: The Identity of a Subject.

UnaryExpression
An UnaryExpression has an UnaryOperator and a single element.

UnaryOperator
UnaryOperator is a type of Operator can take only one element, such as Not.
Instances:
Not

URI
URI (Uniform Resource Identifier).

URL
URL (Uniform Resource Location).

User
Users are Subjects can perform Actions in a system.

Properties:
User_has_Role: User can have Role attribute, which is associated with a set of Permissions.

Validity
Validity is a time period, within which an Attribute or Predicate is valid.
Properties:
validateFrom: The start time of the Validity period.
validateTo: The end time of the Validity period.

Value
Values are used with operators to construct Expressions.

Variable
Variable's value can come from an Expression or properties of other objects.
Properties:
variableName: The name of a Variable.

WithPermission
This predicate states that a Permission must be associated with the Subject of the Request.
Properties:
Assertion_on_Permission: Assert that the request has the Permission to carry out the Action on the Resource.

WithRole
This predicate states that a set of Roles are required to satisfy this predicate. If the User in the Request has the Role asserted by this Predicate or a Role superior to it, then the Precate is TRUE. If there are more than one Role asserted in this Predicate, then user need to have all these roles at the same time.
Properties:
Assertion_on_Role: Asserts that the User has a Role.

11 Trust and Privacy Negotiation

This section introduces and describes the Trust and Privacy Negotiation subsystem of TAS3 and its integration into the K.U.Leuven TAS3 demonstrator. Since this subsystem is based on credential-based access control, where credentials are assumed to be *attribute* credentials that do not necessarily contain unique identifiers (such as distinguished names), the terms "attributes" and "credentials" are used interchangeably in this section. Although attribute credentials are certified attributes, while attributes are not certified, this difference is of no concern in this section. Moreover, it is assumed that each attribute has a *type* (e.g. "employer") and a *value* (e.g. "downtown hospital").

11.1 Introduction

Due to the many different ways one may interpret the words "trust", "privacy", and "negotiation", it is necessary to first define, on a high level, what is meant by these terms. This section (a) explains what is meant by these terms in the context of the TAS3 trust and privacy negotiation subsystem, (b) introduces the purpose of TAS3 trust negotiation, and (c) states two important requirements that the TAS3 trust negotiation subsystem must fulfil.

In TAS3, neither "trust" nor "privacy" is something that can be negotiated; instead, trust is either established or not established based on policies that have been well-specified by data subjects, data owners, service users or service providers that define the types and values of attributes (credentials) that a requesting party must possess in order to access a given resource. Similarly, (an agreement on) privacy is either reached or not reached, as this is defined by a stringent matching process that determines whether or not well-specified privacy requirements (such as data retention periods, acceptable data usages, acceptable recipients of data) of two interacting parties are compatible with each other.

The main purpose of the TAS3 trust negotiation subsystem is to enable a potential client to determine, independently from actually attempting to access a given resource,

(a) whether or not it possesses the right attributes (credentials) in order to access the resource (i.e. to enable the server to establish trust in the client), and
(b) in case where the client indeed possesses the right attributes, a subset of attributes (credentials) that, if disclosed to the server, are sufficient to grant access to the resource.

Examples of these resources include accesses to medical data, employment information, etc. for which the data owner or the data subject has specified the policies that need to be enforced before the corresponding information (and services) can be processed.

A naive approach to solve the problem would be for the client to request, from the server, the policy that governs access to the desired resource. The client could then determine, by examining the policy, which subset of its attributes (if any) suffices to satisfy the policy.

This naive approach, however, neglects two important requirements:

(a) The client may not be willing to indiscriminately disclose its credentials to the server; instead each credential may itself have an associated access control policy that requires the server to prove possession of certain attributes or credentials before the client's credentials may be disclosed to the server.
(b) Servers may not be (and typically are not) willing to expose their policies, at least not in their entirety, to clients. In particular, servers may be willing to expose certain parts of a policy to the general public (e.g. the rule "medical doctors are allowed to access medical files" should be publically accessible), other parts only to clients that possess certain credentials (e.g. the rule "nurses of hospital A are allowed to access summary medical records" should be accessible only to personnel and patients of hospital A), and other parts, such as blacklists, not at all.

The "negotiation" aspect of the TAS3 trust negotiation subsystem arises from requirement (a) above, as follows. Since the client may require *the server* to disclose credentials before the client discloses its own credentials – and these credentials (the ones the client requests from the server) may themselves be protected by further policies that, in turn, require the client to disclose further credentials which may be again protected by access control policies (and so on), a protocol that satisfies requirement (a) would involve the client and the server exchanging policies and credentials in multiple rounds of communication. This exchange would finish either when all exchanged policies have been satisfied by corresponding credential disclosures, or when one party decides that it cannot make further progress either because it does not possess a credential that was requested by the other side, or because it cannot disclose any of the credentials that were requested by the other side (because none of its policies are satisfied).

An interacting party may still choose the exact point in time at which it discloses requested credentials, even when their access control policies have been satisfied. One obvious choice would be to disclose a requested credential as soon as its access control policy has been satisfied by the other party. Another choice would be to hold back an entire set of (requested) credentials until *all* their access control policies have been satisfied. (There exists a multitude of other choices, some of which will become apparent later.) The exact choice of when to disclose which credentials in an ongoing protocol exchange has an impact on both privacy and interoperability[10];

[10] "Interoperability" in this context refers to the ability of two parties to use the protocol in order successfully satisfy each other's policies (i.e. establish trust) whenever their

in fact, there is a trade-off between these two qualities. A party is, in general, free to decide how exactly to strike this trade-off. Striking this trade-off is akin to adopting a *negotiation strategy*. In practice, a (typically configurable) software module takes over this user-centricity task. Such modules are called *negotiation* or *negotiation strategy* modules.

Requirement (a) has an impact on the goal of TAS3 trust negotiation, which is now re-defined as follows. (Requirement (b) has, of course, an impact on the features that the solution has to support; it does not, however, change the overall goal of trust negotiation.) The goal of the TAS3 trust negotiation subsystem is to enable a potential client to determine, independently from actually attempting to access a given resource,

(a) whether or not itself *and the server* possess the right attributes (credentials) in order for the client to access the resource (i.e. to enable the client and the server to establish mutual trust for the purposes of the client accessing the resource),
(b) in case where the client *and the server* indeed possess the right attributes (credentials), a subset of attributes (credentials) that, if the client discloses to the server, are sufficient to grant access to the resource.

11.2 TAS3 trust negotiation concepts

This section describes, on a conceptual level, the policies used for TAS3 trust negotiation as well as the negotiation. Details on policy language, negotiation strategies and further implementation details are given in section 12.

We decided to use the UniPro approach (Winslett, 2003) of automated trust negotiation as the basis for TAS3 trust negotiation because it satisfies the two requirements identified in the previous section and because it is relatively intuitive. We extended and adapted the original abstract approach described in (Winslett, 2003) in several ways in order to fit it into the TAS3 scenario environment.
One of the main features of UniPro is that policies are treated in a manner identical to resources, i.e. policies *are* resources. Resources come with two pieces of metadata: the identifier of the resource itself, and the identifier of the policy that governs access to the resource. Moreover, policies are divided into *fragments*, each of which is treated as a resource in its own right. This allows different parts of a policy to be protected by different policies, thereby enabling the selective disclosure of only parts of a policy to a client. Furthermore, in UniPro, when a given policy fragment is hidden from a client, its *resource identifier* is nevertheless disclosed. Note, however that a single resource may be identified using different resource identifiers, which

---

policies are theoretically compatible, i.e. would allow for this to happen if credentials were disclosed as soon as their respective policies are satisfied.

limits the impact of disclosing this identifier. This serves two important purposes. Firstly, it explicitly enables the other party to determine the presence of a hidden policy fragment, and this is crucial in order to observe the "satisfaction agreement" principle[11]. Secondly, it enables the other party to explicitly refer to the hidden fragment and therefore independently request access to it, or to start a new trust negotiation for it.

In the remainder of this section we will represent UniPro policies as condition trees. Nodes in such trees represent conditions, and are divided into two types: logic conditions and leaf conditions. Leaf conditions may only appear in the leaf positions of the tree, and are predicates over attribute type/value pairs. Logic conditions, which may not appear as leafs, logically connect other conditions together in order to form more complex predicates over attribute type/value pairs, ultimately resulting in the entire access control policy. Each condition (i.e. node in the tree) is treated as a separate resource which may be independently protected by another policy and requested by a client.

As an example, consider the example UniPro access control that protects access to the resource "Johann's Medical Record" (JMR), depicted in Figure 11.1 (left side). Since the root of the tree is a logic "or"-condition with three children, there exist three ways to gain access to JMR: the first way ("Authz. Resr.") is by presenting a credential of a "researcher" that has been authorised by Johann himself to use his medical data for the purposes of research. The second way to obtain access to JMR is by presenting a credential that proves that the requester's name is "Johann" (that is, Johann has access to his own medical record). The third and last way to access JMR is by satisfying all four conditions under the "and"-condition: presenting credentials proving the requestor's status as being a medical doctor ("MD"), his employer certificate ("EMP"), that he is employed by Downtown Hospital ("EMP=DH"), and that his name (or any of the shown attributes/credentials) is not in a blacklist.

---

[11] This principle states that a negotiation protocol should result in an agreement as to the outcome of the negotiation. It should not be possible that one party believes that negotiation completed successfully while the other believes that it failed.

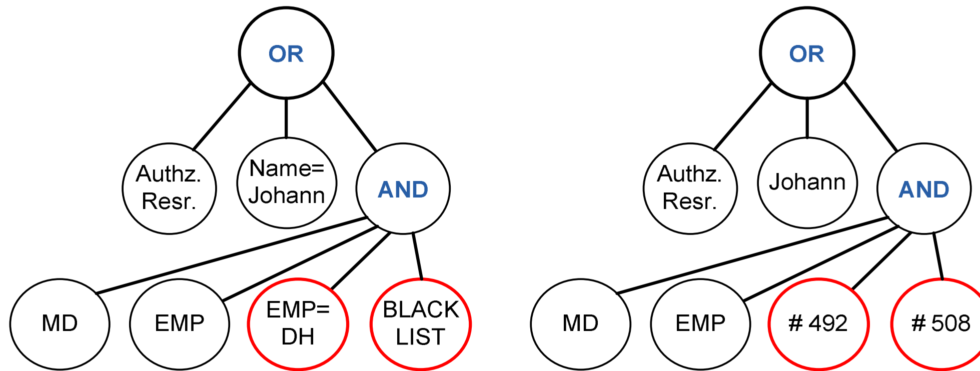


Figure 11.1 - UniPro policy example as a condition tree (left) and applying a filter (right).

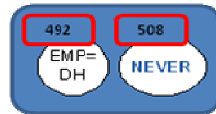


Figure 11.2 - Policies protecting resources #492 and #508

As mentioned earlier, individual conditions may not be freely accessible by the general public, but may themselves be subject to access control. The two red conditions from Figure 11.1 ("EMP=DH" and "BLACKLIST") may be protected "meta"-policies. For example, the "EMP=DH" condition may be protected by a policy that dictates that only employees of Downtown Hospital may gain access to this condition, while the "BLACKLIST" condition may be protected by a "NEVER" policy, which dictates that this resource is never disclosed, no matter what credentials are presented by the client. The two meta-policies are depicted in Figure 11.2, assuming that the resource identifiers of the "EMP=DH" and the "BLACKLIST" conditions are #492 and #508, respectively.

In the following two subsections, we present two example negotiation exchanges (both leading to success) for the resource JMR, which is assumed to be protected by the policy described above. In the first example, the client discloses all required credentials as part of the negotiation while, in the second, it does not. This is to demonstrate that a negotiation can be successful even when the client does not demonstrate to the server that it can satisfy the access control policy of the requested resource.

We stress that the negotiation strategies underlying the following examples, and those employed by the TAS3 demonstrator negotiation strategy module, do *not* correspond to any of the

negotiation strategies defined in (Winslett, 2003); they are instead enhanced version with different properties.

11.2.1 Example negotiation where client reveals all required credentials

Figure 11.3 depicts an example negotiation exchange between a medical doctor (client) that tries to access JMR and the server that hosts this resource. First the client specifies the desired resource (JMR). Then the server sends the access control policy (ACP) for JMR to the client. Note that, since at this point the client has not disclosed any credentials, the ACP for JMR is *filtered* before being sent to the client (see right side of
Figure 11.1). Filtering removes the content of the conditions whose policies are currently not satisfied by the client. Instead only the resource identifiers of these conditions are disclosed.

Since the client does not possess an "Authz Researcher" or an "Name=Johann" credential or attribute, its *negotiation strategy* decides to disclose the required MD credential at this point in time. However, the requested "EMP" credential is protected by a policy, requiring the server to prove its status as a hospital. Hence, the ACP for the "EMP" credential is sent to the server. In the next step, the server discloses its credential proving that it is a hospital, as requested by the client. At that point, the client releases its "EMP" credential, which proves that the medical doctor is employed by Downtown Hospital.

At this point in time, the access control policy of the hidden condition #492 has been satisfied. The server hence pushes the no longer hidden condition to the client. Moreover, assuming that none of the credentials disclosed by the client is in the blacklist, the ACP for JPR is also satisfied at this point in time. Hence, the server responds with a success message, telling the client that the entire access control policy for JMR has now been satisfied.

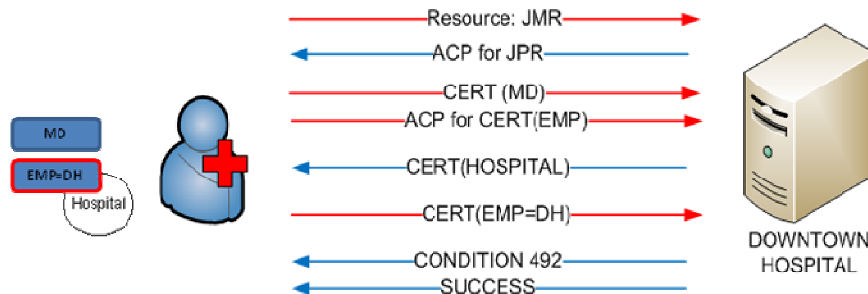


Figure 11.3 - Example negotiation message exchange

11.2.2 Example negotiation where client does not reveal all required credentials

Figure 11.4 depicts another negotiation exchange for the resource JMR, this time initiated by Johann's client. The first two messages are identical with the previous example: the client declares the target resource, and the server responds with the filtered ACP from
Figure 11.1 (right side).
Since Johann's strategy module, which knows that Johann possesses a "Name=Johann" credential, determines that it can satisfy the ACP *beyond any doubt*[12], it sends, at this point, a special "FINISH" message to the server, informing it that it does not wish to continue with the negotiation. The server cannot tell whether or not the client can satisfy JPR's ACP, since the client might have behaved identically, had it determined that it cannot satisfy JPR's ACP. Hence, Johann's privacy is preserved.

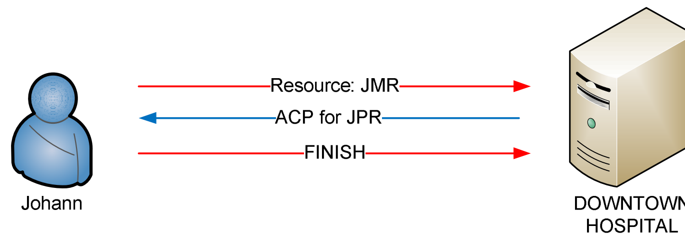


Figure 11.4 - Example negotiation message exchange

Johann's local outcome of the negotiation is, of course, "success", because it has been determined that JPR's ACP can be satisfied by disclosing the "Name=Johann" credential. The server's local outcome is "indeterminate". Note that this, however, does not violate the satisfaction agreement principle: the server does not believe that negotiation failed; instead it does not know the outcome on the client's side.
This example highlights the importance of negotiation strategies: a different strategy may have disclosed the "Name=Johann" credential and waited for the "SUCCESS" message from the server. Moreover, it shows that a successful negotiation (as per the client's view) does not imply the client actually satisfying the ACP of the target resource (by disclosing the necessary credentials); instead it suffices that the client obtains assurance that it *can* satisfy that policy in the future (using the attributes/credentials is already possesses).

11.3 CUP access control policies and trust negotiation

This section describes the implementation of the TAS3 trust negotiation subsystem. Since it is based on the UniPro approach, but has been enhanced and implemented by the COSIC group, we call the resulting system the "COSIC UniPro" (CUP) system. The TAS3 trust negotiation

[12] This can be determined beyond any doubt because Johann's credentials satisfy the ACP in a way that does not assume that any hidden conditions must also be satisfied.

subsystem is the result of the integration of the CUP system into the TAS3 demonstrator; this integration is described in section 12.

The CUP system is essentially a library that extends the TrustBuilder2 (TB2) framework (Adam J. Lee 2009) with an enhanced version of the UniPro approach (Winslett, 2003).

11.3.1 The TrustBuilder2 framework

The TB2 framework is a Java library that provides an API for trust negotiation through iterative disclosure of credentials. It is meant to be extensible in several respects, most notably with respect to

- Policy formats
- Negotiation strategy modules, and
- Credential formats

After extending the TB2 framework in an appropriate fashion, it should be possible that a single TB2 instance is be able to support multiple negotiation strategies, policy and credential formats at the same time. At the beginning of a negotiation exchange, the two interacting TB2 instances parties agree on a “configuration” that specifies in which credential and policy encodings will be used in the remainder of the exchange.

The TB2 system and software architecture fully addresses requirement (a) from section 11.1. This can be seen easily from the example exchange shown in Figure 11.5, where the client (Alice) does not disclose her VISA card credential to the server (Bob) in step 5, until the server satisfies the access control policy for this credential which states that it must disclose a BBB credential first.

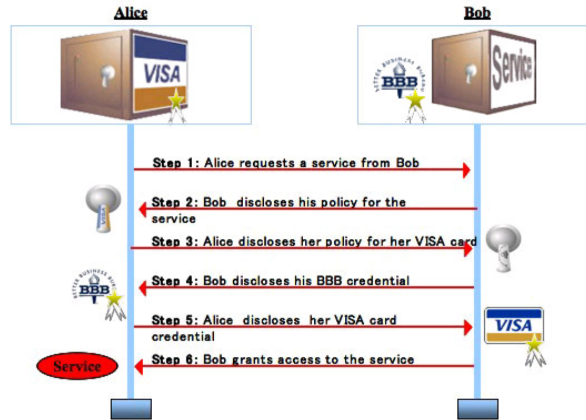


Figure 11.5 - TrustBuilder2 exchange[13]

At the time of writing, TB2 supports a single policy language, namely the Jess policy language. It also supports a "dummy" credential format which resembles X.509 certificates that contain unique identifiers for the subject (such as distinguished names).

We now briefly examine the internal workings of the TB2 system, on a level of abstraction that is appropriate to understand the extensions explained in the next subsection. The exposition here is quite simplified, in order to keep it short and easy to understand. For a more comprehensive exposition of these internal workings the reader is referred to the TB2 manual (Lee 2007) and the software itself (Database and Information Systems Laboratory, Department of Computer Science, University of Illinois at Urbana-Champaign n.d.).

[13] Picture taken from http://dais.cs.illinois.edu/dais/security/trustb.php

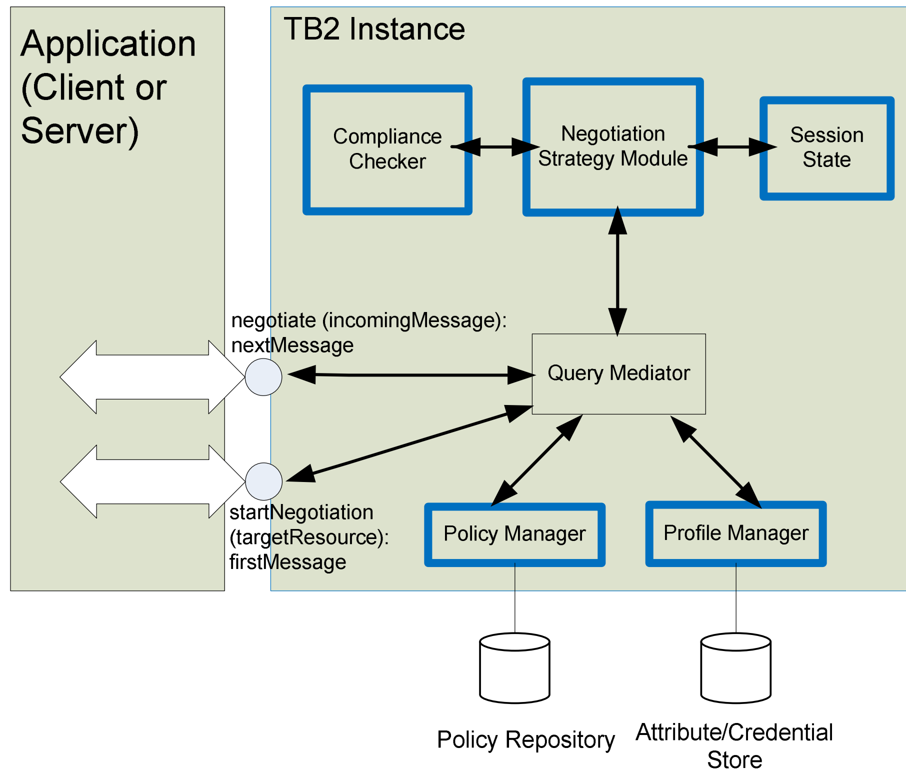


Figure 11.6 - High level overview of internal TB2 structure

Figure 11.6 depicts some of the internal TB2 components, and illustrates the API that TB2 exposes to applications. An application (client or server) that wishes to engage in trust negotiation has to create a TB2 instance. Once created, this instance will offer two main API calls (functions) to the application: **startNegotiation()** and **negotiate()**. If the application is a client it will make use of both functions; a server will typically make use only of the latter.
From the application's point of view, using TB2 is rather simple: **startNegotiation()** expects as the parameter the identifier of the resource ("JMR" in the previous example) that should be the target resource of the negotiation session. The function returns the negotiation message that should be sent to the server and that will initiate the TB2 exchange. Internally, this first message is constructed by the Strategy Module (but it typically trivial since it merely states the target resource)[14].

[14] Before the client sends the target resource identifier to the server, a message exchange takes place in order to establish a common TB2 configuration (policy language,

The **negotiate()** function takes as parameter an incoming negotiation message and returns the next message that should be sent to the other party. Before sending the outgoing message to the other side, the application should, however, examine some of its contents. In particular, it should determine whether or not this is the last message of the session (which can be easily derived from the message). This is because, if it is the last message of the negotiation session, then it should not wait for a reply from the other party. Similarly, before passing an incoming message to the **negotiate()** function, the application should ensure, by examining the incoming message, whether or not an answer is expected from the remote party.

We now briefly explain the tasks of the different components.

Query Mediator: In TB2, different components do not directly invoke each other's functionalities, but instead communicate by means of "Queries" and "Responses", both of which pass through the Query Mediator. This approach is the basis of extensibility, because multiple components can "register" at the query mediator to receive specific types of query, and can react upon such queries. The resolution of which components to send incoming queries from other components, is done during runtime. Several other component types (such a GUI components) are possible with this approach, but are not shown in Figure 11.6.

Profile Manager: The profile manager accepts queries from other TB2 components for locally stored attributes and credentials, and responds to them. During a negotiation, the strategy module typically queries the Profile Manager (through the Query Mediator) in order to check which of the attributes/credentials that are relevant to the ongoing negotiation are actually locally available. The Attribute/Credential store of the TB2 implementation consists of simple text files.

Policy Manager: The policy manager accepts queries from other TB2 components for locally stored authorisation policies. Authorisation policies are referred to by their identifier. If the current TB2 instance is a server, then the profile manager must be able to resolve and provide the policies for both resources and credentials. If it is a client, then the policies in the repository will just cover locally stored credentials. The current TB2 policy manager supports the Jess policy format.

Session state: The session state is an in-memory repository of the current state that is associated with ongoing negotiation sessions. In contains, among other things, a copy of the attributes that are relevant to the current negotiation, a list of data items that have already

---

compatible negotiation strategy family, attribute encodings, etc). This discussion assumes that a common configuration has been established between client and server.

been disclosed to the other side (local policies and credentials), and a list of "holds", i.e. items that were requested but cannot be disclosed because the other side has not yet satisfied associated policies. The session state contains all the information needed by the strategy module in order to fulfil the negotiation strategy.

Compliance Checker: The compliance checker performs a central function in the TB2 framework. It takes as an input an access control policy and a set of attributes (credentials), and it outputs a collection of attribute subsets each of which satisfies the given access control policy. These attribute sets are called 'satisfying sets'. Two types of compliance checkers are envisioned in the TB2 framework: type 1 and type 2. Type 1 compliance checkers output only a single satisfying set (i.e. the collection contains only a single subset), while a type 2 compliance checker output all satisfying (sub)sets. Of course, if no subset of attributes satisfies the policy, then the compliance checker outputs an empty collection.
The strategy module uses the compliance checker in order to find out which subset(s) of locally available attributes (credentials), if any, are sufficient to satisfy any given policy that comes from the remote party. The current TB2 implementation supports a compliance checker for Jess access control policies.

Strategy Module: The strategy module is the heart of the TB2 framework. It builds, given an incoming message, the current session state, and possible outputs from the compliance checker, the next message that should be sent to the remote party. The current TB2 implementation supports a single strategy called the "maximum relevant strategy". This strategy first uses the compliance checker to identify the collection of satisfying sets, then selects one of these credentials sets to disclose to the other party, and then waits until the other party satisfies the policies for all credentials in this set before disclosing any of the credentials. According to this strategy, the client does not check whether or not it has satisfied the server's policy for the target resource during the negotiation, but rather waits for the server to declare whether negotiation has succeeded or failed.

11.3.2 Extending the TB2 framework

Unfortunately, the current TB2 implementation has a number of shortcomings, including the following.

1. It does not address requirement (b) from section 11.1.
2. At the end of a negotiation session, the application merely gets to know whether or not the negotiation was successful. In case of success, the client has no way to query the TB2 instance for the set of credentials that were disclosed during the negotiation, or the set of credentials that satisfy the access control policy of the requested resource (as determined during the negotiation).
3. According to the TB2 negotiation strategy, in order to reach a successful negotiation outcome, the client must disclose to the server a subset of credentials that satisfies the

access control policy of the requested resource. This holds even if it can identify a subset of credentials that satisfies, beyond any doubt, the ACP of the target resource, without actually disclosing the credentials in this set.

4. It does not support “attribute certificates”, i.e. certificates that do not include a unique subject identifier.

In order to address these shortcomings, the modules shown with thick border in Figure 11.6 were extended with the concepts described in section 11.2. The following list describes the new components (CUP stands for “COSIC UniPro”).

CUP Profile Manager: The CUP Profile Manager implements attribute credentials and, similar to the TB2 Profile Manager, uses plain text files for their storage. Credentials are currently not certified

CUP Policy Manager: The CUP Policy Manager supports CUP policies; CUP policies are the COSIC implementation of UniPro policies. The following figure shows the CUP policy encoding of the example policy from
Figure 11.1 (the line numbers are added for easier readability and are not part of the policy format).

```
1: policy_jmr:gdRmwFX+oJYKOIEmaa6i+ljJ; 1

2: KjeidHvHvqDbkgNjAYrSMxjZ: TYPE [MEDICAL_DOCTOR]; 1

3: fO9812kJDKJ3829DJKS831lj: TYPE [EMPLOYER]; 1

4: x4RNJbFOn61Cp0DSLgFLnp7H: BLACKLIST TYPE [NAME] VALUES [Ferdinand Sauerbach]; 0

5: 0GCEzgUZNz8m77YiB4Tchl6r: AND
[KjeidHvHvqDbkgNjAYrSMxjZ,8PDmpldfBj1nbEfRIZkUrSqH,1/xqgxNEYrXUPUW6Cyfuerxz,x4RNJbFOn61Cp0DSLgF
Lnp7H,fO9812kJDKJ3829DJKS831lj]; 1

6: dTKmzVl82uKhhxgNNa3NpEmz: WHITELIST TYPE [NAME] VALUES [Johann]; 1

7: 1/xqgxNEYrXUPUW6Cyfuerxz: TYPE [NAME]; 1

8: gdRmwFX+oJYKOIEmaa6i+ljJ: OR [dTKmzVl82uKhhxgNNa3NpEmz,0GCEzgUZNz8m77YiB4Tchl6r]; 1

9: 8PDmpldfBj1nbEfRIZkUrSqH: WHITELIST TYPE [EMPLOYER] VALUES [DOWNTOWN_HOSPITAL]; onlyemps
```

The first line contains the resource identifier of the current policy (“policy_jmr”), followed by the resource identifier of the root condition of this policy (“gdRmwFX+oJYKOIEmaa6i+ljJ”), and followed by the resource identifier of the policy that protects access to the current policy (“1”). The identifiers “1” and “0” are special identifiers referring to the “ALWAYS” and the “NEVER” policy respectively. The “ALWAYS” policy states that the resource it protects can be disclosed to the general public, while the “NEVER” policy states that the resource it protect is never disclosed.

The remaining lines of the policy may appear in an arbitrary order. The condition representing the root of the tree is defined in line 8. It is a logic condition of type OR and refers to the conditions with resource identifiers dTKmzVl82uKhhxgNNa3NpEmz and 0GCEzgUZNz8m77YiB4Tchl6r. The OR condition is protected by the “ALWAYS” policy (resource identifier “1”) and can hence be disclosed to the general public. The two conditions it refers to are defined in lines 6 and 5 of the policy file. By continuing this analysis the reader will realize that the above file is an encoding of the example from from
Figure 11.1.

The CUP Policy Manager currently understands the following types of condition.
Logic conditions:

- AND: Conditions that requires all children conditions to be satisfied in order to be satisfied itself; may have two or more children conditions.
- OR: Conditions that requires at least one child condition to be satisfied in order to be satisfied itself; may have two or more children conditions.

Leaf Conditions**:**

- WHITELIST: This condition refers to a particular attribute type, and that requires the value of that attribute to be equal to one of the values in a given list of values. For example the condition WHITELIST TYPE [NAME] VALUES [Alice,Bob] requires, in order to be satisfied, the attribute of type “NAME” to have a value of either “Alice” or “Bob”. Otherwise, the condition is not satisfied.
- BLACKLIST: This condition refers to a particular attribute type, and that requires the value of that attribute to *not* be equal to one of the values in a given list of values. For example the condition BLACKLIST TYPE [NAME] VALUES [Eve,Mallory] requires, in order to be satisfied, the attribute of type “NAME” to have any value *except* “Eve” or “Mallory”. Otherwise, the condition is not satisfied.
- TYPE: This condition refers to a particular attribute type, and requires the other part to disclose its attribute of that type. Semantically it is equivalent to a WHITELIST condition where all values are acceptable. (However, since the WHITELIST condition does not support wildcards, adding this condition type was necessary).

CUP Session state: Similar to the TB2 session state, the CUP Session State contains all the information needed by the strategy module in order to fulfil the negotiation strategy. The CUP session state, however, supports enhanced functionality that is necessary for CUP-based negotiation. This includes the dynamic update of policies that contain hidden fragments; the other side may, during a negotiation, decide to disclose certain conditions of an access control policy that were hidden up to that point in time. The session state of the receiver must cope with this situation and must be able to amend the policies accordingly. The CUP session state achieves this.

CUP Compliance Checker: The CUP compliance check is of type 2: given as input a CUP policy with potentially hidden conditions and a set of credentials (attributes), it outputs *all* credential subsets that satisfy the policy. However, due to the fact that certain conditions may be hidden, any given satisfying credential subset may either be satisfying the policy *beyond any*

doubt (if no hidden condition is required to be evaluated is order to satisfy the policy), or may only be *potentially* satisfying (because at least one hidden condition must be evaluated in order to satisfy the policy). The CUP compliance checker therefore outputs *two* collections of satisfying credential subsets: the "surely satisfying sets", and the "potentially satisfying sets". Of course, the compliance checker ensures that the intersection of these two collections is empty.

Note: The TB2 framework does not support this distinction of "surely" and "potentially" satisfying sets. In order to nevertheless support this feature, the framework had to be extended in ways not intended by the TB2 developers.

CUP Strategy Module: The CUP strategy module can cope with CUP policies: whenever previously hidden conditions are revealed by the other side, the affected polices get re-evaluated. The dynamic nature of CUP policies opens up a range of possible negotiation strategies may adopt. The current CUP strategy module is still rather simple, as it operates according to the following principles.

- Hidden conditions are revealed as soon as their access control policies are satisfied. This not only potentially increases the degree of interoperability (since the other side will know more about which credentials to (and which not to) disclose), but also serves as a 'transparency enhancing technology': there is no need to hold back conditions from those who should be able to see them.
- In order to satisfy a policy, the strategy prefers to disclose a "surely" satisfying set over a "potentially" satisfying one. In fact, it chooses the surely satisfying with the lowest cardinality. Only if no surely satisfying sets are available, it chooses a potentially satisfying set (again, the one with the lowest cardinality).
- The strategy does not wait until the policies of the entire selected subset of credentials is satisfied; instead, it discloses individual credentials (from the selected subset) as soon their access control policies are satisfied. The motivation behind this behaviour is to enable the other side to reveal hidden conditions as soon as possible.
- The client immediately stops the negotiation as soon as it identifies a surely satisfying subset of credentials for the "primary" policy, i.e. the access control policy of the target resource.

We extended the TB2 framework in ways not shown in Figure 11.6, for example b y implementing ("dummy") attribute certificates (attribute credentials) against which CUP policies are evaluated. Another important extension arose from addressing the second drawback listed at the beginning of this section. In particular, at the end of a successful negotiation, the TB2 instance could return one of the following data items to the client:

1. A success/failure indication.
2. The collection of attributes (credentials) that led to success i.e. that satisfy the target resource's access control policy.
3. A "ticket" or "token" with which the client can request access to the resource.
4. The target resource itself.

Although Figure 11.5 suggests that the current TB2 system follows option (4), in fact it follows option (1). Our extension follows option (2), for the following reasons:

- Option (1) is not informative enough; the client application should be able to know, after a successful negotiation, which credentials are required in order to gain access to the target resource.
- Options (3) and (4) are not as privacy friendly as option (2): they require, in contrast to option (2), the client to actually disclose all credentials that are required to obtaining access during negotiation. In order words, options (3) and (4) do not allow for the use case described in section 11.2.2.
- Neither option (3) nor option (4) allows negotiation to occur independently from attempting to gain access to the resource: option (3) requires the server component to be able to verify "tickets" that were issued by the server's negotiation component. Option (4) couples negotiation with accessing the resource even more strongly.

11.3.3 Future research directions

The following research questions and directions arose during the implementation of the CUP trust negotiation system.

- What is the effect on interoperability if the negotiation strategy would hold back credentials in a selected subset until all their access control policies are satisfied?
- Since hidden conditions may be revealed during an ongoing negotiation, it may be beneficial for the negotiation strategy to switch the chosen credential subset (the one it decides to disclose to the other party). At other times switching the set may be compulsory (e.g. when a BLACKLIST condition gets revealed that prohibits the use of a certain credential). Based on which criteria should such a switch be made? Possible criteria are the following
    - How large is the intersection of the new candidate set with the set of credentials that have already been disclosed during the negotiation?
    - Is the new candidate set "surely" satisfying while the previously selected set only "potentially" satisfying?
    - Is the newly selected set of lower cardinality that the previously selected set?
- Under which conditions is it beneficial for the client to start independent trust negotiations for hidden conditions before continuing with the current negotiation?
- The negotiation strategy could be augmented with "sensitivity" values of attributes, that affect the selection algorithm of attribute subsets.

12 Integration of CUP trust negotiation into the Prototype

Until version 2.0 of this deliverable, the trust negotiation subsystem as described in section 11 was not integrated. In fact, the tasks of trust negotiation policy definition, evaluation and enforcement were done as described in section 11 above.

For the current version of this deliverable, the trust negotiation, evaluation and enforcement subsystem as described in section 11 has been integrated in the Prototype. This section briefly describes this integration.

12.1 Internal integration

In a given scenario, each actor acts both as a trust negotiation server, and a trust negotiation client. For any given physical machine, each the trust negotiation server of each actor listens to a different TCP port.

12.1.1 Service discovery

Whenever an "edge" is added between two actors, meaning that these actors should be able to directly communicate from that point onwards, a special 'service discovery phase' takes place where the actors exchange messages informing each other which service they have to offer.

These messages where extended to include a new piece of metadata, called the "negotiation metadata" item, which describes the endpoint at which the sending actor accepts trust negotiation sessions. This metadata item contains the IP address and the port number at which the trust negotiation server listens, as well as the identifier of the resource that corresponds to the service being advertised.

12.1.2 Service invocation

The service invocation logic has also been modified. Whenever an actor wishes to invoke a service of another actor, using the metadata it collected during the discovery phase, it first performs trust negotiation for the specific service. As a result of this, the actor will know whether or not the credentials it owns are sufficient to obtain access to the service and, if they are, which subset of its credential will grant access (or potentially grant access, if there exists hidden conditions in the policy).

If sufficient or potentially sufficient credentials are present, then the actor will then go ahead and invoke the service by sending a corresponding message. This message will contain the subset of credentials as this was determined by trust negotiation.

If trust negotiation indicates that the actor does not have sufficient credentials for invoking the service, the service will not be invoked.

12.1.3 Policy enforcement

When receiving a message that triggers a service to be provided, the receiving actor will first check if the subset of attributes/credentials that were included in the message suffices for the satisfaction of the policy that is associated with the requested resource. If they are, then the service is provided, otherwise the message is ignored.

Given that (in the prototype) an actor will not even *attempt* to invoke a service without first having established that the credentials it owns are potentially sufficient, it may seem that enforcing the policy at service time is somewhat redundant.
This, however, is not the case, because the service requestor may not disclose any or all required credential during trust negotiation. In fact, it will stop the negotiation as soon as it can reach a decision as to whether it owns a sufficiently attribute subset or not.

12.2 GUI integration

This section describes the new GUI components that were developed in for the integration of trust negotiation into the prototype.

It is now possible to create CUP policies for every actor in a given scenario. This is done with the policy editor component. Moreover, each actor can be given a number of attribute credentials via the credential editor. Both these components have been newly introduced to the prototype. They can be invoked by right-clicking on any actor, as depicted in Figure 12.1.

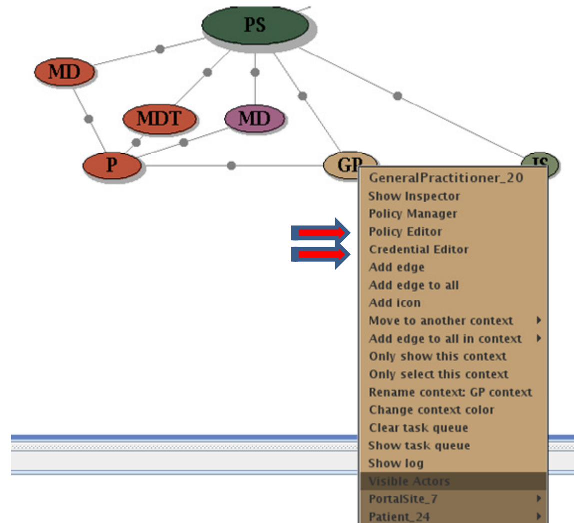


Figure 12.1 Context-sensitive menu (right-click)

12.2.1 CUP Policy Editor

The CUP Policy Editor provides a simple graphical interface for the specification of CUP policies.

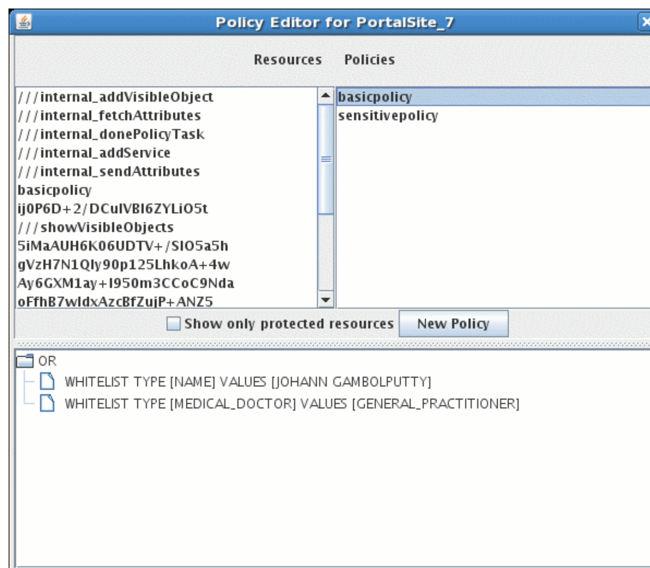


Figure 12.2 - CUP Policy Editor

Figure 12.2 depicts the main CUP policy editor GUI, as shown when invoked via the right-click menu of Figure 12.12.1. On the upper left corner, the resources of the current actor (in the figure "Portalsite_7") are listed. The upper right corner lists the currently known policies for this actor. When the user selects a policy from this list, the lower part shows the policy in tree form (compare to
Figure 11.1, left side). Since the depicted policies are local, they never contain any hidden conditions.

12.2.1.1 Creating a new CUP policy

When the user clicks on the "New Policy" button, the policy editor will first ask the user to assign a policy identifier (such as "basicpolicy") to the new policy he/she is about to create. Then the user is asked to create the root condition of the new policy, via the GUI shown in Figure .

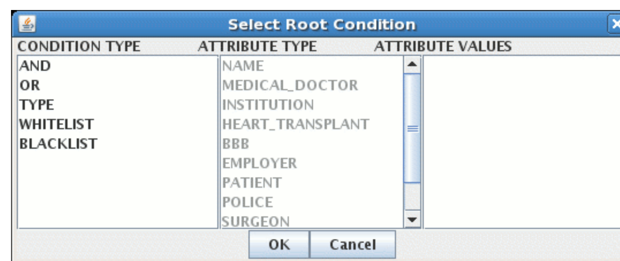


Figure 12.3 - Creating a condition

Depending on the type of condition selected by the user, the attribute type and values field must also be filled in. Logic conditions (AND and OR) do not require the specification of attribute type/value(s). The "type" condition only requires the user to specify the type of the attribute that the condition refers to. Finally, the "whitelist" and "blacklist" conditions require the user to fill in one or more values in the "Values" field before the condition can be accepted.

Once the user has selected the root condition for the new policy, it is immediately added to the list of policies (Figure 12.2). Of course, the policy may not be complete. The user can amend any policy by simply double-clicking on the (logic) condition to which he/she wishes to add a child condition. By doing so, the interface shown in Figure 12.3 will pop up again, and the user can specify a child condition. This can be repeated until the policy is complete.

1.1.1.2 Resources

As described in section 11.2, in the selected trust negotiation approach every policy and even every condition is considered to be a resource. This is why, every time a new policy or condition

is added to the system, the list of resources in the upper left corner of the policy editor is amended with the resource identifiers of the new policy/condition(s).
For policies, the user has to specify these identifiers. However, it would be an unacceptable burden for the user to specify, for each condition, a separate resource identifier. This burden is taken over the system, which generates an random resource identifier for each condition. Care is taken that collisions are avoided.
Because the user does not know, beforehand, what identifier is assigned to individual conditions, the policy editor shows him/her this, as follows. Whenever the user selects a condition in a policy, its identifier is shown in the status bar, and also automatically selected in the resource list. This way the user can easily identify individual conditions that he/she may wish to protect with a policy.

1.1.1.3 Protecting resources with policies

We have described how to create and store new CUP policies for an actor in the prototype. The user may, of course, wish to use these policies to protect resources. This assignment is done very easily: the user first clicks on the identifier of the resource he/she wishes to protect (in the upper left corner). The user then clicks on the identifier of the policy by which he wishes to protect the selected resource. The policy editor will then ask for confirmation of the association (see Figure) and, if confirmed, will store the association. If the resource was already protected by another policy, this association will be overwritten.

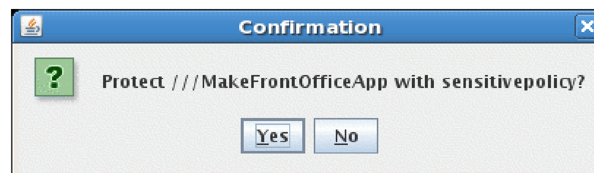


Figure 12.4 - Storing a resource/policy association

The user can, at any time, check if a resource is already protected by a policy, simply by selecting the resource; the status bar will inform him about whether an association exists.

1.1.3 Credential Editor

The credential editor is used to assign credentials to actors. It simulates the real-world situation where actions obtain attribute credential from their respective issuers.

When selecting the corresponding item from the right-click menu shown in Figure 12.12.1, the GUI shown in Figure 72.5 pops up. This GUI informs the user about the credential that the selected actor currently owns. Each credential is a collection of attribute type/value pairs.

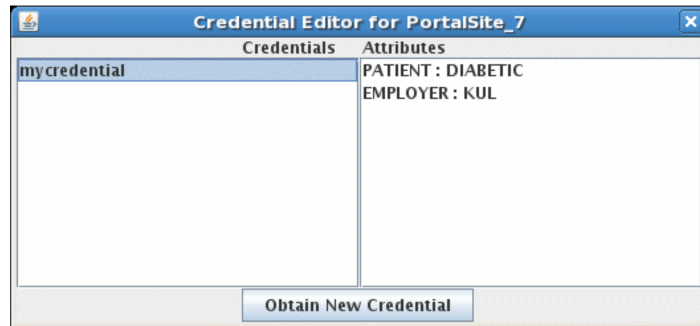


Figure 72.5 - Obtaining credentials

By clicking on the "Obtain New Credential" button, a new credential can be issued to the actor, by means of the GUI shown in Figure 8.6 and Figure 2.7. There, the user should select the attribute types that he wishes to be included in the credential.

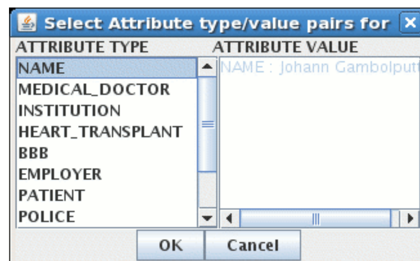


Figure 8.6 - New credential composition

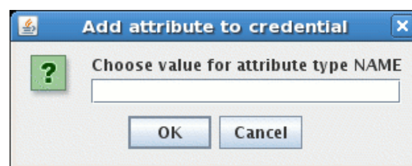


Figure 12.7 - Addition of a new attribute/value pair

When the user selects an attribute type (by double-clicking), the dialog shown in Figure 2.7 pops up, asking the user to specify the value of the attribute for the selected type. The new attribute type/value pair is then added to the credential.

13 Conclusions

This document represents the second phase of the design of an identity management, authentication and authorisation infrastructure for the TAS³ project. This design has been created during the first two years of the project (based in part on the background knowledge and experience of the participants), and in parallel with this design, implementation of some of the functionality of the IdMAA has already been undertaken e.g. we have publicly available demonstrations of Break the Glass policies, delegation of authority and attribute aggregation. Underpinning these is a basic obligation handling infrastructure and credential validation service. Implementation of others features are at various stages of development, and yet others (such as the dynamic management of policies) has not yet been started.

Version 2 of the design shows a significant enhancement over Version 1. We expect that implementation experience and more detailed technical designs will further feed into this design document and will improve it. Integration of the IdMAA infrastructure with the application demonstrators and components from the other work packages will further inform the design. As such there is still significant further work to do in the coming years, and the design will be improved and modified in the light of further implementation experience, integration with the output of the other work packages, and piloting in the application demonstrators.

The following limitations in the previous version have already been addressed:

- we have identified a suitable subscribe/publish messaging system and are currently defining how it will fit into the TAS³ infrastructure. This will be described in deliverable D8.2 [32]
- we have designed the way that sticky policies will be bound to their data as described above

Limitations in the current design that we are aware of are:

1. we are not sure exactly how the policy conflict resolution mechanism will work in the Master PDP;
2. we are not sure how complex it will be to build application dependent PEPs, but we know that we should keep this as simple as possible in order to minimise the work of application developers. Hence the creation of a new conceptual entity, the AIPEP, which can be offloaded with a significant amount of work from the PEP;
3. we have not yet designed the TAS³ protocol for the encapsulating security layer described in section 8, nor the enhanced AIPEP that will support this;

4. we have not designed a language for specifying parameterised obligations
5. we are aware that some obligations might require two phase commit type interfaces and that integrating this with applications, especially legacy ones, could be very problematic;
6. at the time of writing we have not yet integrated the infrastructure described here with work from all of the other WPs, such as secure repositories from WP4 or the event-management bus from WP8 thus these may have an impact on this deliverable. However we have integrated the trust PDP from WP5, the authentication IDP from WP2 and an application demonstrator from WP9, so we are confident that our overall approach is correct.

We are sure that there will also be other limitations that we have not yet documented or are aware of.

We have undertaken the following standardisation activity during the first two years of the TAS^3 project:

1. We have participated in the Open Grid Forum. The editor has chaired the OGSA Authorisation Working Group, and been the editor or co-editor of 4 OGF working documents that have progressed to OGF final specifications in Autumn 2009 [23, 24, 25, 26].
2. Various consortium members are members of the OASIS security services technical committee (SSTC) and the Kantara Initiative (formerly Liberty Alliance project), and have attended various LA meetings throughout the year. In particular, David Chadwick and Sampo Kellomaki attended the LA meeting in Stockholm in July 2008 to present the design for attribute aggregation described in this document, and the mapping of it onto the LA Identity Mapping and Discovery Protocols. They are now progressing changes to the LA Discovery and SAMLv2 Protocols in order to support our attribute aggregation scheme.
3. We have produced a new OASIS draft specification for the dynamic request of attributes [45]
4. The editor is the UK BSI representative to ITU-T X.509 standards meetings. This year the ITU-T group completed the 2009 edition of X.509 and have now started work on the next version of X.500 which will include protocols for password management, which is an important component of identity management.
5. We have been discussing attribute aggregation in Information Cards with Paul Trevithick from the Higgins Project and have attended the Internet Identity Workshop where we led a session on this topic. We also discussed the topic with Michael B Jones of Microsoft, but at the moment he is not willing to commit to this revised model for CardSpace.
6. David Chadwick and Sampo Kellomaki participated in the Service Wave 2009 Future Internet event in Stockholm, Nov 2009.

7. David Chadwick has discussed the carrying of sticky policies in any language with the OASIS XACML TC and they have recently agreed to enhance the XACMLv3 protocol in order to allow any type of policy to be carried, not only XAMCL policies as is specified in the current Committee Draft. This is a major achievement for the TAS3 project. David has also discussed an enhancement for Break the Glass policies, in order to standardise a BTG response in XACML. It has been agreed that a new profile should be written and Seth Proctor from Sun has agreed to collaborate with David on this.

14 Glossary

Application Protocol Enhancement (APE) Model – in this model sticky policies are carried in the application layer protocol along with the application data

Access Control Policy (ACP) – the policy that controls access to a resource. This policy will have different rules depending upon the access control model in use

Attribute Authority (AA) – a source of subject attributes. Can be the source of authority (SoA) or a delegate of the SoA.

Break the Glass (BTG) – a term used to describe an access control policy that allows users who would not normally have access to a resource, to gain access themselves by "breaking the glass" in the full knowledge that they will have to answer for their actions later to their management

Credential Issuing Policy (CIP) – the policy that controls the issuing of credentials by a credential issuing service.

Credential Issuing Service (CIS) – the service that issues digitally signed attribute assertions, provided by an attribute authority.

Credential Validation Policy (CVP) – the policy that controls the validation of credentials by a credential validation service.

Credential Validation Service (CVS) – the service of validating digitally signed attribute assertions and determining which are trusted and which are not, and mapping remotely issued trusted attributes into locally valid ones.

Encapsulating Security Layer (ESL) Model – in this model the application data and sticky policy are transported together in an application independent security protocol (which is still to be defined).

Federated Identity Management (FIM) – The communal services provided by a group of organisations which have set up trust relationships between themselves, so that they can send each other digitally signed attribute assertions about their users' identities in order to grant each others' users access to their resources.

Identity management, authentication and authorization infrastructure (IdMAA) – application independent middleware responsible for authenticating and authorizing entities

Identity Provider (IdP) – an authoritative source of subject attributes (i.e. an AA) that is also capable of authenticating subjects prior to issuing credentials.

Level of Assurance (LoA) – a metric which is used to measure the confidence (or assurance) that a relying party can have, that an authenticated user is really who they say they are. One scale, devised by the US National Institute of Science and Technology, ranges from 1 to 4, with 4 being the highest.

Policy Decision Point (PDP) – the (application independent) part of an access control system that can answer access control requests with a granted or denied decision.

Policy Enforcement Point (PEP) – the (application dependent) part of an access control system that is responsible for enforcing the decisions returned by the PDP.

Personal Identifying Information (PII) – personal information that can be used to identify someone

Privilege Management Infrastructure (PMI) – a highly scalable infrastructure, based on digitally signed attribute assertions, which allows subjects to be authorised to use the resources of relying parties based on their mutual trust in Attribute Authorities. A component of FIM.

Public Key Infrastructure (PKI) – a highly scalable infrastructure, based on public key cryptography, which allows subjects to authenticate to relying parties based on their mutual trust in Public Key Certification Authorities (a type of TTP). A component of FIM.

Separation of Duties (SoD) – a security procedure whereby a high risk task is split into at least two sub-tasks which have to be carried out by different people.

Single Log Off (SLO) – the converse of SSO, whereby a user is simultaneously logged out of all the services that he is currently logged into via SSO.

Single Sign On (SSO) – the process whereby a user can sequentially gain access to a number of computer services by only providing his login credentials once to the first service he contacts.

Source of Authority (SoA) – the ultimate authoritative source for an attribute. A trusted root for authorisation purposes.

Trust Management (TM) – the process of the management of trust between entities. Trust management may rely on many different factors such as the way an entity behaves (behavioural trust), the assertions of trusted third parties, or its performance against a set of trust metrics.

Trust Negotiation (TN) – the process whereby two entities negotiate a trusting relationship between themselves, by sharing their credentials that were issued to them by TTPs that both of them trust.

Trusted Third Party (TTP) – an entity that is trusted by other entities, usually so that the latter may be introduced to each other in order to establish trusted relationships between themselves.

Service Provider (SP) – an entity which offers some kind of electronic service to users.

X.509 - A joint standard by the ITU-T, ISO and IEC which describes both PKI and PMI. X.509 public key certificates are ubiquitously used on the web for SSL/TLS communications with web servers.

15 References

[1] Trusted Architecture for Securely Shared Services, "Annex I - "Description of Work"". 29/11/2007

[2] ITU-T Rec X.812 (1995) | ISO/IEC 10181-3:1996 "Security Frameworks for open systems: Access control framework"

[3] ANSI. "Information technology - Role Based Access Control". ANSI INCITS 359-2004

[4] William E. Burr, Donna F. Dodson, Ray A. Perlner, W. Timothy Polk, Sarbari Gupta, Emad A. Nabbus. "Electronic Authentication Guideline", NIST Special Publication NIST Special Publication 800-63-1, Feb 2008

[5] David W Chadwick, Linying Su, Romain Laborde. "Coordinating Access Control in Grid Services". Concurrency and Computation: Practice and Experience, Volume 20, Issue 9, Pages 1071-1094, 25 June 2008.

[6] OASIS. "XACML v3.0 Obligation Families Version 1.0" Working draft 3. 28 December 2007

[7] R. L. "Bob" Morgan, Scott Cantor, Steven Carmody, Walter Hoehn, and Ken Klingenstein. "Federated Security: The Shibboleth Approach". Educause Quarterly. Volume 27, Number 4, 2004

[8] Arun Nanda. "Identity Selector Interoperability Profile v1.0" Microsoft Corporation, April 2007. see http://download.microsoft.com/download/1/1/a/11ac6505-e4c0-4e05-987c-6f1d31855cd2/Identity-Selector-Interop-Profile-v1.pdf

[9] David Chadwick, George Inman, Nate Klingenstein. "Authorisation using Attributes from Multiple Authorities – A Study of Requirements". Presented at HCSIT Summit - ePortfolio International Conference,16-19 October 2007, Maastricht, The Netherlands.

[10] Hodges, J. Cahill, C.(Editors). "Liberty ID-WSF Discovery Service Specification V2.0". Liberty Alliance Project

[11] OASIS. "Assertions and Protocol for the OASIS Security Assertion Markup Language (SAML) V2.0", OASIS Standard, 15 March 2005

[12] OASIS. "Level of Assurance Authentication Context Profiles for SAML 2.0". Working Draft 1. 1 July 2008

[13] Hodges, J. Aarts, R. Madsen, P. and Cantor, S.(Editors). "Liberty ID-WSF Authentication, Single Sign-On, and Identity Mapping Services Specification v2.0". Liberty Alliance Project.

[14] Hodges, J. Kemp, J. Aarts, R. Whitehead, G. Madsen, P. "Liberty ID-WSF SOAP Binding Specification v2.0" Liberty Alliance Project.

[15] OASIS "eXtensible Access Control Markup Language (XACML) Version 2.0"
OASIS Standard, 1 Feb 2005

[16] D.W.Chadwick. "Dynamic Delegation of Authority in Web Services" in "Securing Web Services: Practical Usage of Standards and Specifications". Edited by Dr Panayiotis Periorellis, Newcastle University. Idea Group Inc. 2008. pp111-137 ISBN 978-1-59904-639-6. Information about the book can be found at http://www.igi-global.com/reference/details.asp?id=6976

[17] ISO 9594-8/ITU-T Rec. X.509 (2001) "The Directory: Public-key and attribute certificate frameworks"
[18] O. Bandmann, M. Dam, and B. Sadighi Firozabadi."Constrained delegation". In Proceedings of the IEEE Symposium on Research in Security and Privacy, pages131-140, Oakland, CA, May 2002. IEEE Computer Society Press.
[19] Mont, M.C.; Pearson, S.; Bramhall, P. "Towards accountable management of identity and privacy: sticky policies and enforceable tracing services". Proc 14th Int Workshop on Database and Expert Systems Applications, 1-5 Sept. 2003. Page(s): 377 – 382
[20] B. Ramsdell et al. "Secure/Multipurpose Internet Mail Extensions (S/MIME) Version 3.1 Message Specification". RFC 3851. July 2004
[21] Bertino, E., Ferrari, E., Squicciarini, A.: Trust Negotiations: Concepts, Systems and Languages. IEEE Computer, pp. 27-34, 2004.
[22] Wensheng Xu, David Chadwick, Sassa Otenko. "A PKI based secure audit web service". IASTED Communications, Network and Information Security CNIS, November 14 - November 16, 2005, Phoenix, USA
[23] V. Venturi, T. Scavo, D.W. Chadwick, "Use of SAML to retrieve Authorization Credentials", GFD. 158. 13 November 2009. Available from http://www.ogf.org/documents/GFD.158.pdf
[24] David Chadwick, Linying Su. "Use of WS-TRUST and SAML to access a Credential Validation Service". GFD.157. 13 November 2009. Available from http://www.ogf.org/documents/GFD.157.pdf
[25] David W Chadwick, Linying Su, Romain Laborde. "Use of XACML Request Context to Obtain an Authorisation Decision". GFD.159. 13 November 2009. Available from http://www.ogf.org/documents/GFD.159.pdf
[26] David Chadwick. "Functional Components of Grid Service Provider Authorisation Service Middleware". GFD. 156. 29 October 2009. Available from http://www.ogf.org/documents/GFD.156.pdf
[27] TAS3 Architecture, TAS3 Deliverable D2.1, Version 1.0. May 2009
[28] D.W.Chadwick, A. Otenko. "RBAC Policies in XML for X.509 Based Privilege Management" in Security in the Information Society: Visions and Perspectives: IFIP TC11 17th Int. Conf. On Information Security (SEC2002), May 7-9, 2002, Cairo, Egypt. Ed. by M. A. Ghonaimy, M. T. El-Hadidi, H.K.Aslan, Kluwer Academic Publishers, pp 39-53.
[29] W3C. "The Platform for Privacy Preferences 1.0 (P3P1.0) Specification" available at http://www.w3.org/TR/P3P/ (accessed 24 October 2008)
[30] Ninghui Li, John C. Mitchell, William H. Winsborough. "Design of a Role-based Trust-management Framework". IEEE Symposium on Security and Privacy, Oakland, May 2002
[31] Gansen Zhao, David Chadwick, and Sassa Otenko. "Obligation for Role Based Access Control". *IEEE International Symposium on Security in Networks and Distributed Systems (SSNDS07)*. 2007.
[32] TAS3 Deliverable D8.2. "Software Documentation: System: Back Office Services", Version 2.0, due month 39
[33] TAS3 Deliverable D4.3. "Integrated Secure Repositories", Version 1.0.

[34] OASIS. "SAML 2.0 Profile of XACML, Version 2.0". Committee Draft 1, 16 April 2009
[35] N. Zhang, L. Yao, A. Nenadic, J. Chin, C. Goble, A. Rector, D. Chadwick, S. Otenko and Q. Shi; “Achieving Fine-grained Access Control in Virtual Organisations”, Concurrency and Computation: Practice and Experience, John Wiley & Sons Ltd. Vol. 19, Issue 9, June 2007, pp. 1333-1352
[36] S. Tuecke, V. Welch, D. Engert, L. Pearlman, M. Thompson. “Internet X.509 Public Key Infrastructure (PKI) Proxy Certificate Profile”. RFC3820, June 2004.
[37] David W Chadwick, Sassa Otenko and Tuan Anh Nguyen. “Adding Support to XACML for Multi-Domain User to User Dynamic Delegation of Authority”. International Journal of Information Security. Volume 8, Number 2 / April, 2009 pp 137-152. DOI 10.1007/s10207-008-0073-y
[38] O. Bandmann, M. Dam, and B. Sadighi Firozabadi. “Constrained delegation”. In Proceedings of the IEEE Symposium on Research in Security and Privacy, pages131-140, Oakland, CA, May 2002. IEEE Computer Society Press.
[39] Ninghui Li, William H. Winsborough, John C. Mitchell. “Distributed credential chain discovery in trust management”.Journal of Computer Security 11 (2003) pp 35–86
[40] D.W.Chadwick, S. Anthony. “Using WebDAV for Improved Certificate Revocation and Publication”. In LCNS 4582, “Public Key Infrastructure. Proc of 4th European PKI Workshop, June, 2007, Palma de Mallorca, Spain. pp 265-279
[41] Dwaine Clarke, Jean-Emile Elien, Carl Ellison, Matt Fredette, Alexander Morcos, Ronald L. Rivest. “Certificate chain discovery in SPKI/SDSI”. Journal of Computer Security, Issue: Volume 9, Number 4 / 2001, Pages: 285 - 322
[42] Y. Elley, A. Anderson, S. Hanna, S. Mullan, R. Perlman and S. Proctor, “Building certificate paths: Forward vs. reverse”. *Proceedings of the 2001 Network and Distributed System Security Symposium (NDSS’01)*, Internet Society, February 2001, pp. 153–160.
[43] Housley, R., Ford, W., Polk, W., and Solo, D., "Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile," RFC 3280, April 2002
[44] Log4J Manual, obtainable from http://logging.apache.org/log4j/1.2/manual.html
[45] G.Inman, D.W.Chadwick. “OASIS Draft. SAMLv2.0 SSO Extension for Dynamically Choosing Attribute Values”, November 2009
[46] “TAS3 Protocols and Concrete Architecture” Deliverable D2.4. 31 December 2009
[47] OASIS “Profiles for the OASIS Security Assertion Markup Language (SAML) V2.0”, OASIS Standard, 15 March 2005
[48] “TAS3 Common Ontologies”. Deliverable D2.2, Version 1.8, 22 May 2009.
[49] “TAS3 Lower Common Ontology”. Deliverable D2.3, Version 0.6, Dec 2009

Appendix 1. The Obligation Schema and Java Interface

The TAS3 obligation enforcement infrastructure is obligation policy language independent. Applications can specify their obligation policies in any language they choose providing that it is wrapped in an XML wrapper conforming to the XACML obligation schema defined in [15] and reproduced in A1.1 below. An example of how an obligation, written in the Simple Obligation Language specified in D2.1, needs to be wrapped in order to be carried by the TAS3 authorisation infrastructure, is given in section A1.2 below. The Java interface for an Obligation Service that will enforce such obligations in the TAS3 authorization infrastructure reference implementation is provided in section A1.3 below.

A1.1 Obligation Policy Wrapper Schema

The following schema for an Obligation is copied from the XACML standard [15]

```
<xs:element name="Obligation" type="xacml:ObligationType"/>
<xs:complexType name="ObligationType">
  <xs:sequence>
          <xs:element ref="xacml:AttributeAssignment" minOccurs=”0”
          maxOccurs="unbounded"/>
  </xs:sequence>
<xs:attribute name="ObligationId" type="xs:anyURI" use="required"/>
<xs:attribute name="FulfillOn" type="xacml:EffectType" use="required"/>
</xs:complexType>
```

The schema for AttributeAssignment and AttributeValue are also copied from [15] and are given below.

```
<xs:element name="AttributeAssignment"
type="xacml:AttributeAssignmentType"/>
<xs:complexType name="AttributeAssignmentType" mixed="true">
  <xs:complexContent>
    <xs:extension base="xacml:AttributeValueType">
      <xs:attribute name="AttributeId" type="xs:anyURI" use="required"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

```
<xs:element name="AttributeValue" type="xacml:AttributeValueType"
substitutionGroup="xacml:Expression"/>
<xs:complexType name="AttributeValueType" mixed="true"> <xs:complexContent>
 <xs:extension base="xacml:ExpressionType">
  <xs:sequence>
        <xs:any namespace="##any" processContents="lax" minOccurs="0"
        maxOccurs="unbounded"/>
  </xs:sequence>
  <xs:attribute name="DataType" type="xs:anyURI" use="required"/>
  <xs:anyAttribute namespace="##any" processContents="lax"/>
 </xs:extension>
</xs:complexContent>
</xs:complexType>
```

Only one attribute assignment is needed, this being the obligationDescription assignment which will hold the obligation policy as a string. We define this below:

```
<AttributeAssignment>
        AttributeId="urn:?:?:TAS3:attribute:obligationDescription"
        DataType="http://www.w3.org/2001/XMLSchema#string">
</AttributeAssignment>
```

A1.2 Example Obligation Policy

The following example wraps an obligation written in the TAS3 Simple Obligation Language (SOL) inside the standard XACML obligation schema.

```
<Obligation ObligationId=http://TAS3.eu/TAS3sol/PrivacyPurpose
FulfillOn="Permit">
        <AttributeAssignment> AttributeId= "urn:?:?:TAS3:attribute:obligationDescription"
        DataType="http://www.w3.org/2001/XMLSchema#string">
                urn:?:?:TAS3:sol:vers=1
                urn:?:?:TAS3:sol:delon=1255555377
                urn:?:?:TAS3:sol1:use=urn:TAS3:sol1:use:purpose
```

```
                        urn:?:?:TAS3:sol:share=urn:TAS3:sol1:share:group
                        urn:?:?:TAS3:sol1:repuse=urn:TAS3:sol1:repuse:opr
    </AttributeAssignment>
</Obligation>
```

A1.3 Java Interface

Table 1: The ObligationInformation interface

```
/**
 * Interface to obligation descriptions. Each {@code Obligation}
 * can tell its identifier and can return its actual description
 *
 * @author Stijn Lievens
 * @version 0.1
 */
public interface ObligationInformation {

	/**
	 * Returns an identifier characterizing the kind of obligation.
	 *
	 * @return a String representation of the Obligation identifier
	 */
	String getObligationId();

	/**
	 * Returns the actual object representation of the obligation.
	 * This could be an {@code Element} or an {@code ObligationType}
	 * or a {@code String}, etc. We do not limit the representation
	 * types.
	 *
	 * @return an Object representing the description of the obligation
	 */
	Object getObligationContent();


	/**
	 * Returns the fallback obligations when the original obligation
	 * fails. Should never be null but might be empty if there is no
	 * fallback position.
	 *
	 * @return the list of fallback positions for this obligation
	 */
	List<ObligationInformation> getFallbackPosition();
}
```

Table 2: The ObligationConstructor interface

```
/**
 * Interface giving a method to construct a {@code Obligation}
 * from an {@code ObligationInformation} and a context.
 *
 * @author Stijn Lievens
 * @version 0.1
 */
public interface ObligationConstructor {

        /**
         * Creates an {@code Obligation} from the given {@code ObligationInformation}.
         * Implementations should try to catch as much problems here so as to maximize
         * the chances of the {@code doObligation} method succeeding later on.
         * At the very least it should be checked that the obligation information
         * is complete and correctly specified.
         *
         * @param obligation a description of the obligation
         * @param context the context (e.g. the XACML request context) that
         * the returned obligation will operate on
         * @return an Obligation that will execute the actions specified
         * by the obligation upon calling {@code doObligation}
         */
        Obligation construct(ObligationInformation obligation, Object context)
          throws ObligationConstructorException;

        /**
         * Returns the obligation identifiers of the obligations it knows how
         * to handle.
         *
         * @return a Collection containing the known obligation identifiers
         */
        Collection<String> getObligationIds();
}
```

Table 3: The Obligation interface

```
/**
 * Interface for a task (obligation) that can be executed.
 *
 * @author Stijn Lievens
 * @version 0.1
 */
public interface Obligation {

        /**
         * This method should perform the task specified by obligation.
         * Implementations should try to throw as little exceptions as
         * possible and make sure (as far as possible) that the Obligation
         * will succeed when it is first constructed.
         *
         * Implementations should take care to free any resources they have allocated
         * after the obligation has finished.
         *
         * @throws Exception when something goes wrong with the execution
         * @see ObligationConstructor
         */
        void doObligation() throws Exception;

        /**
         * This method should be called to free any resources taken up by
         * the {@code Obligation}. It is an error to call {@code doObligation}
         * after {@code freeResources} has been called.
         */
        void freeResources();
}
```

The following table lists the two most important methods from the ObligationsService class.

Table 4: Most important methods of ObligationsService class

```
/**
 * Class that coordinates the execution of obligations.
 * Its main method is the {@code doObligations} method that will
 * try to build {@code Obligation} objects from the
 * {@code ObligationInformation} objects and will then try to execute
 * these obligations.
 *
 * @author Stijn Lievens
 * @version 0.1
 */
public class ObligationsService {
     /**
          * Adds an {@code ObligationConstructor} under the given identifier.
          * After calling this method, the {@code ObligationInformation} objects
          * that return an id equal to the one set here will be constructed using
          * the passed in {@code ObligationConstructor}.
          *
          * @param id the identifier of the obligations that will be constructed
          * with the given {@code ObligationConstructor}
          * @param obConstructor the {@code ObligationConstructor} that will be used
          * to construct {@code Obligation} objects from {@code ObligationInformation}
          * objects that return the given id
          */
         public void addObligationConstructor(String id,
             ObligationConstructor obConstructor) {
                   implementation omitted  }
     /**
          * Performs the actual obligations.
          *
          * @param obligations the list of {@code ObligationInformation} objects
      * that need to be enforced
          * @param context the request context that the obligations should use
          * @return the list of unrecognised obligations when the object is not built in
          * the 'processAll' mode. Otherwise the list should always be empty.
          * @throws ObligationsServiceException when something goes wrong
          */
         public List<ObligationInformation> doObligations(
                 List<ObligationInformation> obligations,
                         Object context) throws ObligationsServiceException;
```

Appendix 2 The CVS Policy Schema

```
<?xml version="1.0" encoding="utf-8"?>
<xs:schema elementFormDefault="qualified" xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:annotation>
    <xs:documentation>
      PERMIS Policy CVS Schema
      Author: Dmitry Bragin [db91@kent.ac.uk]
      Editors: David Chadwick, Sassa Otenko
      Semantic checks that can't be performed by the WXS are also documented. Search for "Semantic:"
    </xs:documentation>
  </xs:annotation>
<!-- 2008-05-22: sfl, added ID attribute on MSoDPolicy element
                                    changed 'ActionsRef' into 'ActionRef'
                        added an optional TimeZone attritute on the root element
v51     2009-11-06:dwc changed description of role element, added optional role mapping policy

        -->
  <xs:simpleType name="OID">
    <xs:annotation>
      <xs:documentation>
        This is the definition of OIDd as used in RoleSpecs
      </xs:documentation>
    </xs:annotation>
    <xs:restriction base="xs:string">
      <xs:pattern value="\d+[(\\.\d+)]*"/>
    </xs:restriction>
  </xs:simpleType>

  <xs:complexType name="TimeType">
    <xs:annotation>
      <xs:documentation>
        This is the definition of date-time format as used in permis. For compatibility with the old policies
        this is declared as a string in SimpleTimeType rather that xs:dateTime
      </xs:documentation>
    </xs:annotation>
    <xs:attribute name="Time" type="SimpleTimeType" />
  </xs:complexType>

  <xs:simpleType name="SimpleTimeType" >
    <xs:annotation>
      <xs:documentation>
        This is the definition of date-time format as used in permis. For compatibility with the old policies
        this is declared as a string rather that xs:dateTime
```

```
    </xs:documentation>
  </xs:annotation>
  <xs:restriction base="xs:string"/>
</xs:simpleType>

<xs:complexType name="DomainSpecType">
  <xs:sequence minOccurs="1" maxOccurs="unbounded">
    <xs:element ref="Include" />
  </xs:sequence>
</xs:complexType>

<xs:annotation>
  <xs:documentation>
    What follows is a definition of the IF statement grammar.
  </xs:documentation>
</xs:annotation>

<xs:element name="PERMIS CVS_Policy">
  <xs:annotation>
    <xs:documentation>
      Semantic: WXS (W3C XML Schema Language) doesn't have any way of specifying the root element of
a document,
      so the code has to check whether this really is the root element
    </xs:documentation>
  </xs:annotation>
  <xs:complexType>
    <xs:all>
      <xs:element ref="SubjectPolicy" />
      <xs:element ref="RoleHierarchyPolicy" />
      <xs:element ref="SOAPolicy" />
      <xs:element ref="RoleAssignmentPolicy" />
      <xs:element minOccurs="0" ref="RoleMappingPolicy" />
    </xs:all>

    <xs:attribute name="OID" type="xs:string" use="required" >
      <xs:annotation>
        <xs:documentation>
          Semantic: Software has to check it this is either an OID or URN and display a warning if it isn't
        </xs:documentation>
      </xs:annotation>
    </xs:attribute>

            <!-- start changes by sfl -->
            <xs:attribute name="TimeZone" type="xs:string" use="optional" >
            <xs:annotation>
                        <xs:documentation>
```

```
                    The TimeZone attribute specifies the TimeZone to use in the policy.
                    If it is absent, or equals the empty string, then local time will be used.
                    If present, it should be a TimeZone as recognised by Java.
                    Semantic: software should check that it is indeed a valid time zone.
                </xs:documentation>
            </xs:annotation>
            </xs:attribute>
            <!-- end changes by sfl -->
  </xs:complexType>

  <!--Declarations of reference constraints follow-->

  <!--@Type on a SubjectDomain in RoleAssignmentPolicy/RoleAssignment has to refer to @ID on
  a declared SubjectDomainSpec-->
  <xs:keyref name="SubjectDomainSpecRef" refer="SubjectDomainSpecKey">
    <xs:selector xpath="./RoleAssignmentPolicy/RoleAssignment/SubjectDomain"/>
    <xs:field xpath="@ID"/>
  </xs:keyref>

  <!--@Type on a role in RoleAssignmentPolicy/RoleAssignment/RoleList has to refer to @Type on a
previously declared RoleSpec-->
  <xs:keyref name="RoleSpecRef2" refer="RoleSpecKey">
    <xs:selector xpath="./RoleAssignmentPolicy/RoleAssignment/RoleList/Role"/>
    <xs:field xpath="@Type"/>
  </xs:keyref>

  <!--@ID on a SOA in RoleAssignmentPolicy/RoleAssignment has to reference @ID of an exisitng
SOASpec-->
  <xs:keyref name="SOASpecRef" refer="SOASpecKey">
    <xs:selector xpath="./RoleAssignmentPolicy/RoleAssignment/SOA"/>
    <xs:field xpath="@ID"/>
  </xs:keyref>

  <!--@Value on a SubRole under RoleHierarchyPolicy/RoleSpec/SupRole/SubRole has to refer to @Value of
a SupRole-->
  <xs:keyref name="SupRoleRef" refer="SupRoleKey">
    <xs:selector xpath="./RoleHierarchyPolicy/RoleSpec/SupRole/SubRole"/>
    <xs:field xpath="@Value"/>
  </xs:keyref>
 </xs:element>

 <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"/>

  <xs:element name="RoleHierarchyPolicy">
    <xs:complexType>
      <xs:sequence minOccurs="1" maxOccurs="unbounded">
        <xs:element ref="RoleSpec" />
      </xs:sequence>
    </xs:complexType>

    <xs:key name="RoleSpecKey">
      <xs:selector xpath="./RoleSpec"/>
      <xs:field xpath="@Type"/>
    </xs:key>

    <xs:key name="SupRoleKey">
      <xs:annotation>
        <xs:documentation>Please note that this key has to be defined on this element as scope of a keyref
cannot refer to its parent element</xs:documentation>
      </xs:annotation>
      <xs:selector xpath="./RoleSpec/SupRole"/>
      <xs:field xpath="@Value"/>
    </xs:key>
```

```
  <xs:unique name="OIDUnique">
   <xs:selector xpath="./RoleSpec/SupRole"/>
   <xs:field xpath="@OID"/>
  </xs:unique>
</xs:element>

<xs:element name="RoleSpec">
  <xs:complexType>
   <xs:sequence minOccurs="1" maxOccurs="unbounded">
    <xs:element ref="SupRole" />
   </xs:sequence>
   <xs:attribute name="Type" type="xs:string" use="required">
    <xs:annotation>
     <xs:documentation>
      RoleSpec type is a string, typically the LDAP attribute type name for the
      attribute in the role assignment AC.
      RoleSpec OID is the object identifier of the attribute type in the role
      assignment AC
     </xs:documentation>
    </xs:annotation>
   </xs:attribute>
   <xs:attribute name="OID" type="OID" use="required" />
  </xs:complexType>
</xs:element>

<xs:element name="SupRole">
  <xs:complexType>
   <xs:sequence minOccurs="0" maxOccurs="unbounded">
    <xs:element ref="SubRole" />
   </xs:sequence>
   <xs:attribute name="Value" type="xs:string" use="required">
    <xs:annotation>
     <xs:documentation>
      Value is the attribute value of the SupRole. We have currently restricted
      the value to be an XML identifier, so that it can be referred to in other
      parts of the policy (for example as a SubRole). This restricts the role
      attribute syntax to be a PrintableString. (If this proves to be too
      restrictive we can replace ID by CDATA in a subsequent version of the
      policy.)
      Note that the key for the @Value is declared at the RoleHierarchyPolicy element
     </xs:documentation>
    </xs:annotation>
   </xs:attribute>
  </xs:complexType>
  <xs:unique name="SupRoleValueUnique">
   <xs:selector xpath="SupRole"/>
```

```
    <xs:field xpath="@Value"/>
    </xs:unique>
  </xs:element>

  <xs:element name="SubRole">
    <xs:complexType>
      <xs:attribute name="Value" type="xs:string" use="required">
        <xs:annotation>
          <xs:documentation>
            Value is a reference to a SupRole value defined elsewhere within this
            RoleSpec. Corresponding keyref element is located in the root element of the schema.
          </xs:documentation>
        </xs:annotation>
      </xs:attribute>
    </xs:complexType>
  </xs:element>

  <xs:element name="SOAPolicy">
    <xs:annotation>
      <xs:documentation>
        The SOA Policy contains security parameters of trusted SOAs. These are
        administrators who are trusted to issue attribute certificates/assertions. At the moment
        we do not need any parameters except the SOA's LDAP DN (or a plain DN) or URL. Both comma (,)
separated and slash (/) separated formats are supported. CN=guest,OU=GlobusTest,O=Grid means the
same as /CN=guest/OU=GlobusTest/O=Grid.
        Later we may want
        to specify Cross Certification or Recognition of Authority rules here e.g. how to map policies and how
to map
        external roles into internal roles of this security domain.
        There must be at least one SOASpec, otherwise no attribute assertions will be trusted. If the policy
writer is going to issue
        attribute certificates, then his/her name should be here. Otherwise this can be delegated to the other
trusted SOAs in the SOA Policy.
      </xs:documentation>
    </xs:annotation>
    <xs:complexType>
      <xs:sequence maxOccurs="unbounded">
        <xs:element ref="SOASpec" />
      </xs:sequence>
    </xs:complexType>
    <xs:key name="SOASpecKey">
      <xs:selector xpath="./SOASpec"/>
      <xs:field xpath="@ID"/>
    </xs:key>
  </xs:element>
```

```
  <xs:element name="SOASpec">
    <xs:complexType>
      <xs:attribute name="ID" type="xs:string" use="required">
        <xs:annotation>
          <xs:documentation>
            The ID is a valid XML ID for reference to this SOA anywhere in this
            policy. This is now also done via a key defined below.
          </xs:documentation>
        </xs:annotation>
      </xs:attribute>
      <xs:attribute name="LDAPDN" type="xs:string">
        <xs:annotation>
          <xs:documentation>
            Semantic: WXS doesn't have enough power to reliably represent a DN grammar, so the software
should verify the validity of this attribute
          </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:complexType>
  </xs:element>

  <xs:element name="RoleAssignmentPolicy">
    <xs:complexType>
      <xs:sequence minOccurs="1" maxOccurs="unbounded">
        <xs:element ref="RoleAssignment" />
      </xs:sequence>
    </xs:complexType>
  </xs:element>

  <xs:element name="RoleAssignment">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="SubjectDomain" />
        <xs:element ref="RoleList" />
        <xs:element ref="Delegate" />
        <xs:element ref="SOA" />
        <xs:element ref="Validity" />
      </xs:sequence>
      <xs:attribute name="ID" type="xs:string" />
```

```
  </xs:complexType>
</xs:element>

<xs:element name="SubjectDomain">
  <xs:complexType>
    <xs:attribute name="ID" type="xs:string" use="required">
      <xs:annotation>
        <xs:documentation>
          See the root element declaration for the appropriate keyref
        </xs:documentation>
      </xs:annotation>
    </xs:attribute>
  </xs:complexType>
</xs:element>

<xs:element name="RoleList">
  <xs:complexType>
    <xs:sequence>
      <xs:element minOccurs="0" maxOccurs="unbounded" ref="Role" />
    </xs:sequence>
  </xs:complexType>
</xs:element>

<xs:element name="Role">
  <xs:annotation>
    <xs:documentation>
      If Role Type is missing, it means any role type and value.
            It is not permitted to have a specific role value without a role type.
      If Role Type is present and Role Value is missing, it means any value of the Role Type.
      If both are present then it means only this specific role type and value,
            and any subordinate values of the same type from the role hierarchy.
      Data types for both attributes have been changed to xs:string from IDREF
    </xs:documentation>
  </xs:annotation>
  <xs:complexType>
    <xs:attribute name="Type" type="xs:string" />
    <xs:attribute name="Value" type="xs:string" />
  </xs:complexType>
</xs:element>

<xs:element name="SOA">
  <xs:complexType>
    <xs:attribute name="ID" type="xs:string" use="required">
      <xs:annotation>
        <xs:documentation>
          See the root element declaration for the appropriate keyref
```

```
        </xs:documentation>
      </xs:annotation>
    </xs:attribute>
  </xs:complexType>
</xs:element>

<xs:element name="Validity">
  <xs:annotation>
    <xs:documentation>
      The RoleAssignmentPolicy Validity time serves to restrict the validity
      time of issued role assignment ACs, and to discard ACs that are too old, or are
      outside the
      bounds of the maximum and minimum validity periods. The actual validity time
      is the intersection of the policy absolute validity time and the AC validity
      time.
      The Age sub-element specifies the maximum age of an AC, relative to the
      evaluation time. If the AC notBefore validity time precedes the Age, it will be
      discarded.The Maximum and Minimum sub-elements specify maximum and
      minimum periods, relative to the evaluation time, that an AC must be valid
      for, in order for it to be accepted
    </xs:documentation>
  </xs:annotation>
  <xs:complexType>
    <xs:sequence>
      <xs:element minOccurs="0" name="Absolute">
        <xs:complexType>
          <xs:attribute name="Start" type="SimpleTimeType" />
          <xs:attribute name="End" type="SimpleTimeType" />
        </xs:complexType>
      </xs:element>
      <xs:element minOccurs="0" name="Age" type="TimeType"/>
      <xs:element minOccurs="0" name="Maximum"  type="TimeType"/>
      <xs:element minOccurs="0" name="Minimum"  type="TimeType"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>

<xs:element name="Delegate">
  <xs:complexType>
    <xs:attribute name="Depth">
      <xs:annotation>
        <xs:documentation>
          Depth  is an integer that specifies the level of delegation that is
          allowed
          0 means no delegation is allowed (SOA-&gt;user direct)
          1 means 1 level of delegation is allowed (SOA-&gt;AA-&gt;user) etc.
```

```
        if depth is missing infinite delegation is allowed
      </xs:documentation>
    </xs:annotation>
    <xs:simpleType>
      <xs:restriction base="xs:integer">
        <xs:minInclusive value="0"/>
      </xs:restriction>
    </xs:simpleType>
  </xs:attribute>
 </xs:complexType>
</xs:element>

<xs:element name="RoleMappingPolicy">
 <xs:complexType>
  <xs:sequence>
   <xs:element minOccurs="1" maxOccurs="unbounded" ref="RoleMappingRule" />
  </xs:sequence>
 </xs:complexType>
</xs:element>

<xs:element name="RoleMappingRule">
 <xs:complexType>
  <xs:sequence>
   <xs:element name=externalRole type="Role" />
   <xs:element name=internalRole  type="Role" />
  </xs:sequence>
 </xs:complexType>
</xs:element>

</xs:schema>
```

Appendix 3. A Remote Obligation Enforcement Scenario

A user has a privacy policy on his electronic medical record which says that any consultant surgeon can read his EMR on the condition that if they make a remote copy they must delete it two weeks after they have finished treating him.

Assume that the patient with national health ID 123 is referred to the consultant Mr Knife and the latter is assigned to treat the patient in the hospital's work allocation system. This is done by the assignment of the parameterised role TreatingPatient to Mr Knife, where the role value is the patient's health ID i.e. TreatingPatient=ID 123.

Mr Knife authenticates to the EMR system and requests to retrieve the EMR for patient ID 123. The application asks the CVS to retrieve Mr Knife's credentials and it obtains Role=Doctor issued by the national registrar for doctors, Employer=MidSec Hospital issued by the hospital, and TreatingPatient=ID 123 issued by the hospital's work allocation system. These are validated by the CVS as being issued by their respective authoritative sources. The application then asks the AIPEP/Master PDP if the user can be granted access to the EMR of patient ID 123, passing it the user's valid attributes. The AIPEP/Master PDP returns granted with a "before" obligation that the PEP should stick this embedded policy to the data. This embedded policy says that Mr Knife should delete the record within two weeks of his TreatingPatient credential being revoked[15].

Assuming Mr Knife is accessing the EMR system from a remote location and wants to take a copy of the record, then the embedded obligation policy cannot be enforced by the EMR system since it is remote from the workstation that Mr Knife is using to retrieve the EMR. The obligation enforcement must be carried out by the workstation receiving the EMR, which means that the patient's EMR must be carried along with a sticky policy (containing the embedded obligation and any other privacy policies of the user) to the workstation. This is why the local obligation is simply one to stick the embedded policy to the data.

This raises a number of issues

i) What is the format for the data and its sticky policy
ii) Which component creates this data element
iii) How is it carried to the remote workstation
iv) Which component at the remote workstation receives this data element and unpacks and unsticks it
v) Which component at the remote workstation ensures obligation enforcement is carried out sometime in the future.

The proposed answers to the above are as follows:

i) We have proposed an application independent StickyPAD data structure/schema for data and its sticky policy. We propose that this data structure is used to carry the

[15] If the access is being done locally and Mr Knife is not making a copy of the record, then the obligation can be safely ignored. If the access is being done remotely and a copy will be made then the obligation will need to be enforced. It is possible to make this decision within the Master PDP if the location of Mr Knife (with a value of remote or local) is passed as an environmental attribute to the PDP and the policy has separate rules for local and remote access. Otherwise the PEP will need to know whether to ignore the obligation or not depending upon its destination. Either way the PEP will have some extra work to do concerning the location of Mr Knife.

EMR and sticky policy if the application does not have its own application dependent way of doing it.

ii) We propose that the PEP is the only component that can create the StickyPAD or application dependent message since it is retrieves the data from its local EMR database and the authorisation infrastructure never sees the entire EMR data. However, a general purpose obligation service can be built for producing stickyPADs, the CreateStickyPAD obligation which can probably be based on S/MIME.

iii) Once the PEP has created the StickyPAD or application dependent message it transfers this to the remote workstation using the existing application retrieval protocol.

iv) The PEP in the remote workstation receives the incoming message and unpacks and unsticks it.

v) The PEP calls the AIPEP/MasterPDP passing it the sticky policy and asking if Mr Knife is entitled to receive the EMR for this patient. The AIPEP asks the Master PDP, and if a granted reply is received it will contain a “with” obligation to delete the created EMR file within 2 weeks of his TreatingPatient credential being revoked. The PEP will call the “futureDeleteFile” obligation simultaneously with creating the local EMR file. If the obligation fails, then Mr Knife is refused permission to save this EMR to the local filestore. If the obligation succeeds, then the EMR file is created along with an obligation which has already been recorded in secure stable storage and an associated event handling system to ensure the file will be deleted within two weeks of the TreatingPatient credential being revoked. When the “TreatingPatient” credential is revoked, this sends an event to the event handler, which calls the futureDeleteFile obligation. This records in secure stable storage that the TreatingPatient credential has been revoked and it sets an event that in two weeks from now it should be woken up to check that the file has now been deleted. In two weeks time the timer event fires and calls the futureDeleteFile obligation that checks if the file has been deleted, and if not, it deletes it. The obligation is then complete and is removed from stable storage.

Document Control

Amendment History

Version	Date	Comments
0.1	13 Dec 2008	Initial version by David Chadwick and Lei Lei Shi
0.2	16 Dec 2008	Commented version from Marc Santos
0.3	22 Dec 2008	Updated ontology model from Lei Lei Shi
1.0	3 January 2009	Included all comments from internal reviews, and added missing sections such as Glossary and Exec Summary
1.1	12 jan	Edits on ontology. Updated TOC
1.2	26 April 2009	Edits to address external reviews comments

1.8	7 Nov 2009	Major rewrite (initial draft) to add new features and align with D2.1
1.9.n	1-31 Dec 2009	Versions which addressed all the comments from the internal reviews
2.0	2 January 2010	V2 released to public
2.1	12 Jan 2010	Trust negotiation chapters added