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:	7 December 2010
Version:	3.0

SEVENTH FRAMEWORK PROGRAMME

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	KIT	DE	KARL	Partner
20	Symlabs SA	PT	SYM	Partner

Contributors

	Name	Organisation
1	David Chadwick, Lei Lei Shi, Stijn Lievens, Ana Ferreira, Kaniz Fatema, George Inman, Tom Luu	University of Kent
2	Sampo Kellomaki	Symlabs
3	Danny de Cock, Andreas Pashalidis	KU Leuven
4	Marc Santos	University of Koblenz-Landau
5	Ioana Ciuciu	VUB

Table of Contents

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. Sections 11 and 12 describe the trust and privacy negotiation mechanism and how this is integrated into the TAS^3 architecture. Section 13 concludes by describing the current limitations in the design, and indicating where future research is still needed. Section 13 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 TAS^3 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 final version 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 aligned with the architecture deliverable D2.1 [27] but drills down into more detail. 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. *Although this is the final iteration of the design of an extremely complex authorization infrastructure that has never been created before, it may still contain bugs, flaws, or omissions. The reader should be aware that even though this is the final version of the design document, not all of the designed components have yet been fully implemented. Consequently the finally delivered open source software (D7.2) in month 48 may deviate from this design in some ways. Evaluation of the IdMAA infrastructure is not included in this design document.*

1.1 Research Contributions

The specific research contributions of WP7 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. Two new conceptual and functional component, a backend Linking Service and a front end Linking Identity Selector, are proposed, whose purpose is to link together a user's different Identity Provider accounts and attributes;
- 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;
- wrote a number of application independent obligation handling services that can be easily integrated into applications via the standard obligations interface and standard XACML obligation policy statements;
- 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;

- integrated a delegation token issuing service (DIS) into a standard IdP, to provide delegation of authority between users who may or may not be known to the IdP or the DIS
- devised and implemented a credentials and policies negotiation service that incrementally discloses policies, policy fragments, and attribute credentials between a client and a server, in order to establish knowledge of access control prerequisites without compromising privacy;
- devised a conceptual and functional component, the ontology-based interoperation service (OBIS), whose role is to semantically match the security attributes of the service requester to the security policy attributes of the service provider and to return the domination relation between them. This linguistic-based ontology approach is adopted in the privacy domain in order to allow different non-technical users (and organizations) to express their security policies in an intelligible way, through the use of natural language, thus enforcing the TAS3 user-centricity aspect.
-

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 TAS^3 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 provides 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. Sections 11 and 12 describe the trust and privacy negotiation mechanism and how this is integrated into the TAS^3 architecture. Section 13 concludes this deliverable, by describing the current limitations in the design, indicating where further research still needs to be done in the future. 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 no two individuals within a system can have the same identifier. However, this identifier is only

[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

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[3]. Hence systems such as Shibboleth [1], SAMLv2 [11] and CardSpace [2] were born. TAS^3 uses this model for authentication and authorisation.

[3] Note however that the lack of a correlation handle does not necessarily in itself preserve the user's privacy. If the set of identity attributes known to each SP are sufficient to uniquely identify users (e.g. email address or name and postal address) then the SPs may still collude and correlate the user's information.

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 necessarily 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 for the IdP to inform 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[4].

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 TAS^3 has enhanced the ISO ABAC model so that subject attributes may be 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 that issues 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

[4] In a TAS^3 application level protocol 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 the TAS^3 authorisation protocol, the target is always specified in the XACML request context.

resource owner (called the Target Source of Authority in Figure 2.1) specifies the credential validation policy which directs the CVS on how to control this 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 refer to a subject's roles, we mean any attributes that have been assigned to or asserted by 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 TAS^3 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 TAS^3 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 this case 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 assurance.

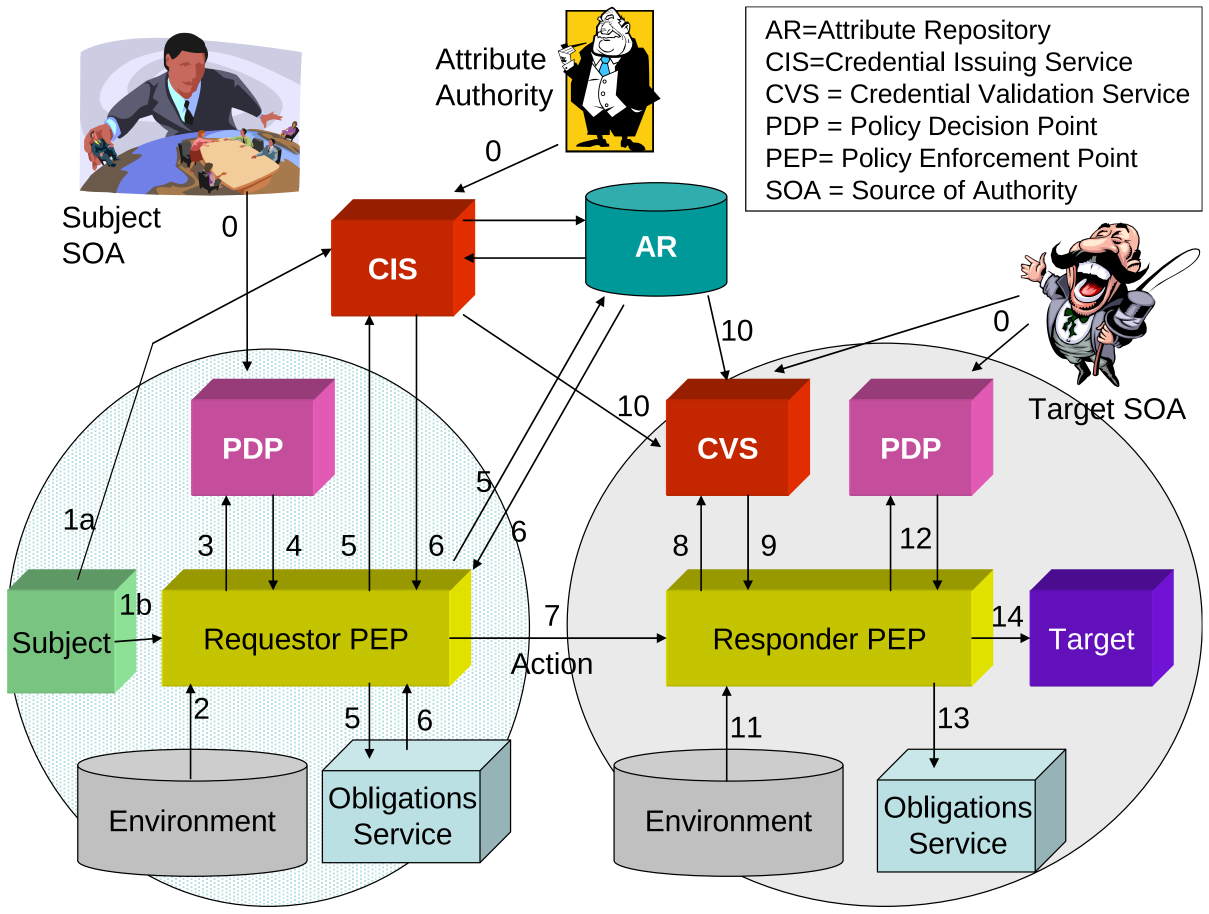


Figure 2.1. Request Authorization Model

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, 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 TAS^3 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 TAS^3 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 TAS^3 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.

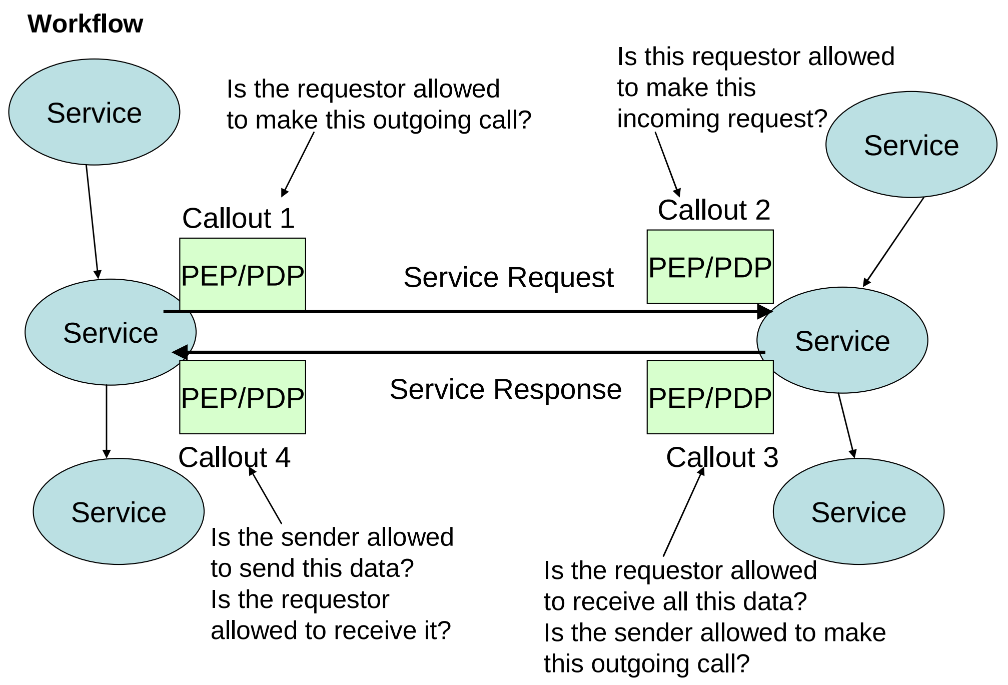

Workflow
Service
Is the requestor allowed to make this outgoing call?
Is this requestor allowed to make this incoming request?
Service
Callout 1
Callout 2
PEP/PDP
Service Request
PEP/PDP
Service
Service
PEP/PDP
Service Response
PEP/PDP
Callout 4
Callout 3
Service
Service
Is the sender allowed to send this data? Is the requestor allowed to receive it?
Is the requestor allowed to receive all this data? Is the sender allowed to make this outgoing call?

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 PEP. 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.

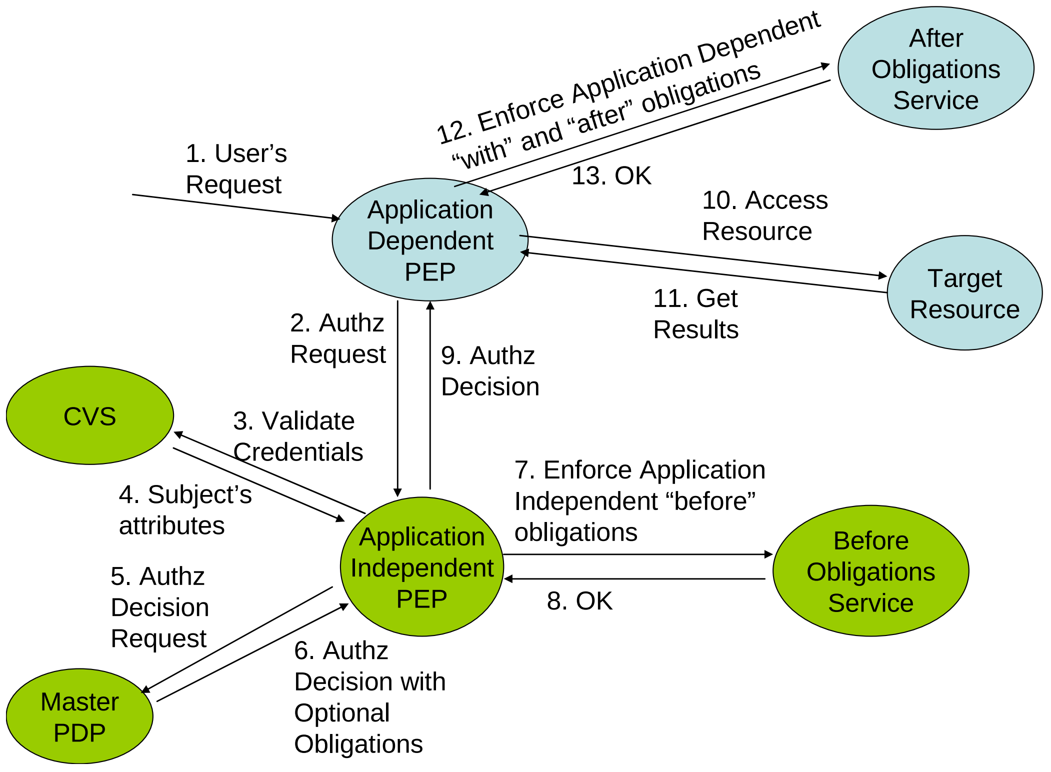

After Obligations Service
12. Enforce Application Dependent "with" and "after" obligations
13. OK
1. User's Request
Application Dependent PEP
10. Access Resource
Target Resource
11. Get Results
2. Authz Request
9. Authz Decision
CVS
3. Validate Credentials
4. Subject's attributes
7. Enforce Application Independent "before" obligations
Application Independent PEP
Before Obligations Service
8. OK
5. Authz Decision Request
6. Authz Decision with Optional Obligations
Master PDP

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 TAS^3 has contributed 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 and has not been addressed in the TAS^3 project.

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 type is enforced by a specific Obligation Handling Service, which is registered with the Obligations Service at program initialisation. Each Obligation Handling 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 TAS^3 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 handling service object that enforces it. The obligation policy itself can be carried around the TAS^3 infrastructure along with the other sticky policies and eventually it will end up being passed to the correct obligation handling 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.

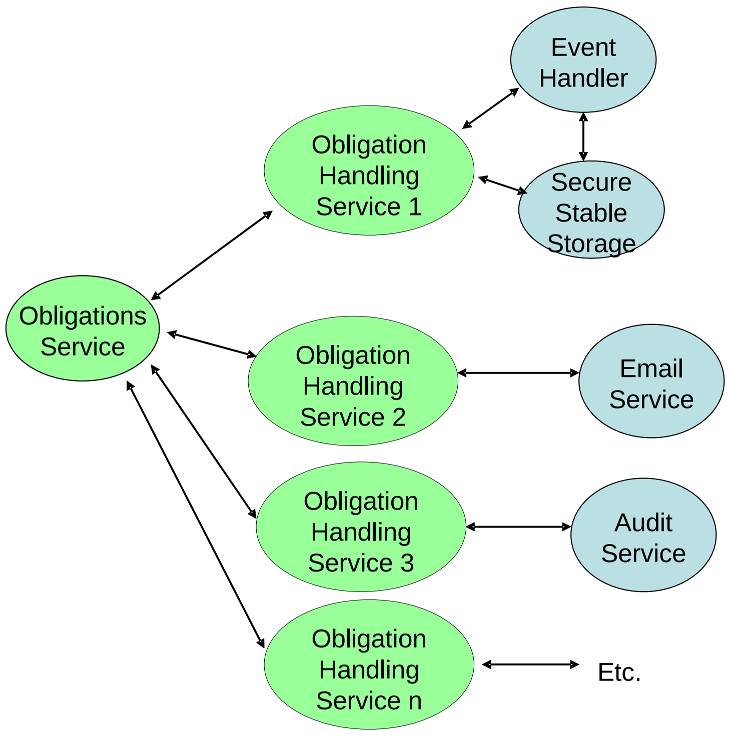

Event Handler
Obligation Handling Service 1
Secure Stable Storage
Obligations Service
Obligation Handling Service 2
Email Service
Obligation Handling Service 3
Audit Service
Obligation Handling Service n
Etc.

Figure 2.4. The Obligations Service

In the TAS³ project we have provided a number of general purpose Obligation Handling Services that can be used by many different applications, such as: an Email obligation handling service that can be used to notify a person about an event, an auditControl obligation handling service that will adapt the amount of auditing information that is recorded (see section 3.3), and a delete file obligation handling 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 have used 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 TAS³ 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.

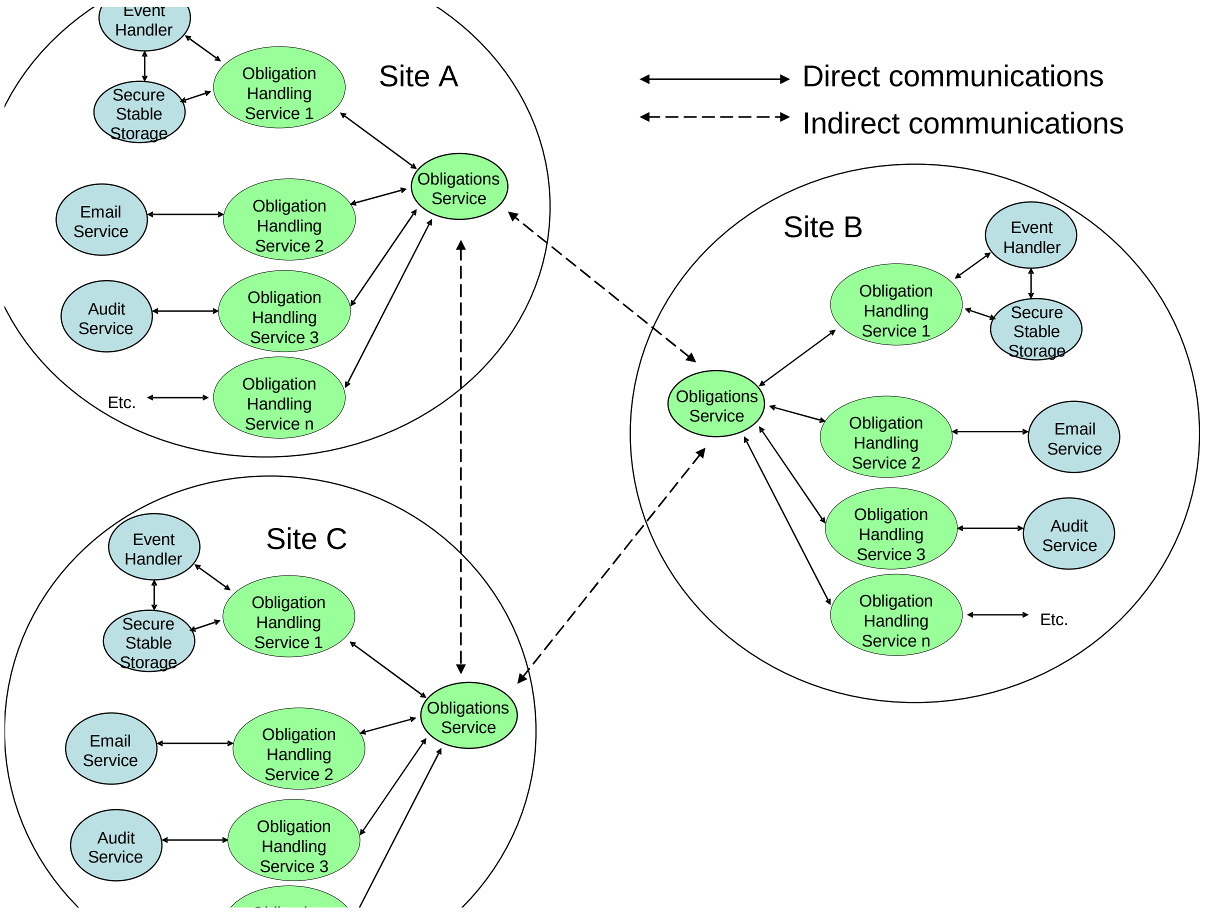


Figure 2.5. Distributed Handling of Obligations

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 Handling Service Interfaces

Each Obligation Handling 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 ***getObligationId*** 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). When this method returns 'null', this means that this obligation does not care which Obligations Service it is executed by. Consequently the first Obligations Service that recognises this obligation will enforce it.
- 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. When this method returns 'null', this means that the obligation does not have an explicit temporal type associated with it. Consequently the first Obligations Service that recognises this obligation will enforce it.
- 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 ***getFallBackPosition*** 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 is 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 e.g. the To, From and Subject fields of an email obligation. 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 ObligationConstructor 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 an ObligationConstructorException when the Obligation can't be created, for example when: the ObligationConstructor doesn't recognise the obligation type, there is some missing information in the request context, 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 Exception 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 handling services, and ensuring that, as far as is possible, they either all successfully complete or none do. As stated above, one of the reasons for constructing all of the obligation handling objects before executing any of them, is to limit the possibilities of failure partway through. If all obligation handling 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 (note that this feature in not yet implemented).

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. This allows the ObligationsService 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. If the subject URN returned by the ObligationInformation is null then the ObligationsService will execute the Obligation if it recognises it.

The ObligationsService object comprises amongst others the following 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 ObligationConstructor to use 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 exactly or the subject URN of the ObligationInformation is null, 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 when the ObligationInformation's subject URN is not null throw an exception "unrecognised obligation" along with the obligation ID. When the ObligationInformation's subject URN is null then put the ObligationInformation in the to-be-returned list of un-enforced obligations.

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 getFallbackPosition 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 getFallbackPosition 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 TAS^3 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 related policy rule which says the same users can access a resource when the particular BTGi state is TRUE, 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. In the implementation P^{BTG} is encoded in the XACML response context as a Deny response with a newly defined BTG status code. We have submitted this specification to the XACML TC for standardisation.

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.

For pure BTG policies, class B) and class T) users will always be the same group of people. However, our implementation is flexible enough that 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 (though it can be argued that this is no longer BTG access, but is rather *manager controlled* access). 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.

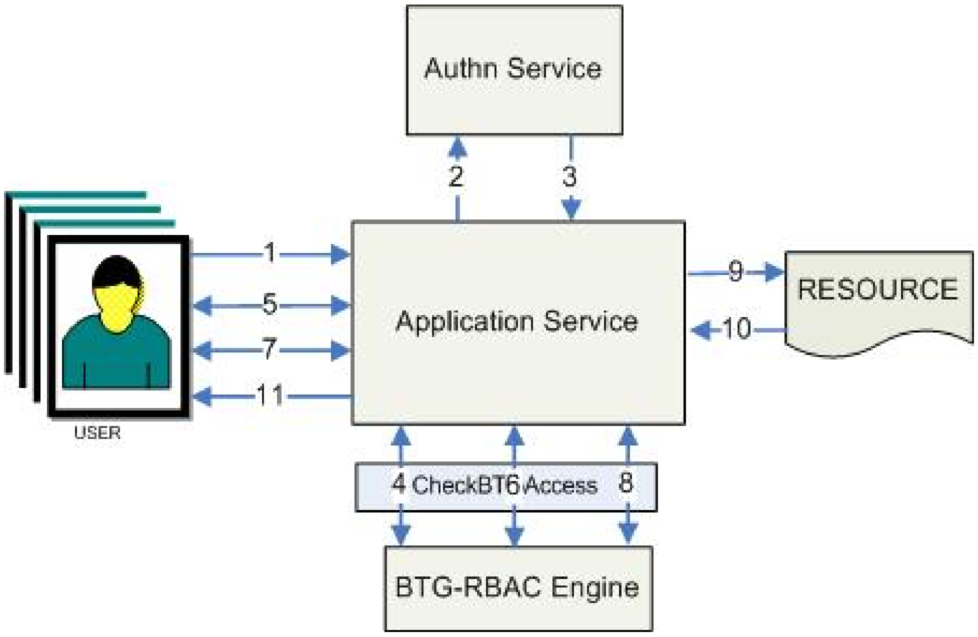


Figure 3.1. Using the Break the Glass PDP

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. 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 *r2* is allowed to "break the glass" for *reading obs1* but the system must perform three obligations if *r2* does this, and Role *r3* 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	$reset^{BTG}$	BTGi		

3.1 The BTGi state variable

Conceptually, the BTG-state of the system is determined by a collection of BTGi state variables. Each BTG variable is either TRUE or FALSE at any point in time. Each BTG-variable has a certain name, examples are: BTG:(subject-id(S),action-id(A))[5] and BTG:(role(S)=Doctor,resource-id(R)). In general a BTG variable name is made up of a set of attribute names (or attributes for short) and/or a set of attribute names with fixed values (or attribute values for short), as described below:

BTG:(SubjectDimensions, ActionDimensions, ResourceDimensions, EnvDimensions)

where SubjectDimensions is a (possibly empty) list of strings where each string refers to a set of subjects characterized by the particular attribute or attribute value. An empty string implies all subjects. Analogous statements hold for ActionDimensions, ResourceDimensions and EnvDimensions. The notation we use to refer to these attributes is

attributeName(Object)[=value],

where Object is one of S, A, R or E (for Subject, Action, Resource and Environment respectively). In the policy only BTG-variable names are mentioned, and the fact that a permission assignment is dependent upon the BTG-state is done by referencing a BTG variable name and imposing a condition on it e.g. IF BTG:(role(S),action-id(A)) .EQ. TRUE. All BTG-obligations associated with the BTG operation will reference the same variable name and require it

[5] Note that in actual policies the names of the attributes used in the BTG variable name must be the same as the names of the attributes that are passed in the request context so that they can be matched for equality. So if the subject-id attribute is passed as urn:oasis:names:tc:xacml:1.0:subject:subject-id in the request context, then this is the name that must be used in the BTG variable name, which is rather unwieldy to use in examples.

(implicitly) to be set to TRUE. All BTG-obligations associated with the ResetBreakTheGlass operation (see section 4.6) will reference the same BTG variable name and require it (implicitly) to be set to FALSE.

So, the BTG-state of the system is described in a fashion very similar to the use of coordination attributes in [5]. In order to better understand how the BTG-state works, it may help to see how the BTG-state could be stored and managed. Conceptually, the BTG-state consists of a number of BTG-state tables, each table labelled by its variable name. If a variable name is dependent upon *n* attributes, then conceptually the table has *n+1* columns: one for each attribute together with an additional column to hold the actual value of the variable instance represented by the row. Any attribute values in the BTG variable do not require columns in the table since their values are fixed. So, consider the BTG variable name BTG:(role(S)=doctor,subject-id(S),action-id(A)), and suppose that a subject named 'Hani' has broken the glass for the action 'GET' and a second subject named 'Gareth' has broken the glass for the action POST. This would give rise to the following table:

TABLE 3.2 BTG:(ROLE(S)=DOCTOR,SUBJECT-ID(S),ACTION-ID(A))

subject-id(S)	**action-id(A)**	**value**
Hani	GET	TRUE
Gareth	POST	TRUE

Note that FALSE values for every other subject-id/action-id combination don't actually need to be stored in the table as FALSE is the default value for all BTG variable instances.
The BTG-state holds a number of these tables (as configured when created), so for instance it could hold a second table for BTG:(role(S),resource-id(R),action-id(A)=GET) like this:

TABLE 3.3: BTG:(ROLE(S),RESOURCE-ID(R),ACTION-ID(A)=GET)

role(S)	**resource-id(R)**	**value**
Research Associate	CN=Record123,O=UKC, C=GB	TRUE

This would be the resulting table after a subject in possession of the role Research Associate has broken the glass on the GET action for the resource with distinguished name CN=Record123, O=UKC, C=GB.

3.1.1 Resetting the BTGi State Variables

There is a need to reset BTG variables at some point in time. This can be done automatically, semi-automatically or manually. Automatic resetting means that the BTG-RBAC engine itself resets the BTG variable to FALSE after a specified event has occurred. The event must be specified by the policy writer when creating the BTG-RBAC policy. Semi-automatic and manual resetting of the BTG variable are initiated in a standardised way by either a system component or a human being that is external to the BTG-RBAC engine. For this we specify new reset actions for a BTG variable table and instance. The policy writer must specify who (which roles) has permission to perform either of these actions and which BTG variables it will affect.

Our implementation supports all three modes for resetting BTG variables, using the same underlying technique. Specifically, we use the existing obligations service that is internal to the BTG-PGP, and define two new obligations, namely: ResetBreakTheGlassObligation and ResetBreakTheGlassTableObligation. The former resets a BTG variable, whereas the latter resets an entire BTG table.

Automatic resetting requires the policy writer to specify the ResetBreakTheGlassObligation when defining the corresponding BreakTheGlassObligation. Both obligations should be associated with the permission to break the glass. Our obligation implementation currently supports a single event type, namely the elapsed time, which ensures that a BTG variable is reset when a specified time interval has elapsed. Thus, say that the BTG variable BTG:(subject-id(S),action-id(A)) needs to be reset 30 minutes after it has been set. This can be achieved with the XACML obligation shown in Figure 3.2. The policy writer should combine this with the obligation to break the glass and associate both of these with the appropriate permission to break the glass. When the obligations are enforced, the values for any attribute names in the BTG variable (subject-id(S) and action-id(A) in this example) are resolved from the same request context so that the same BTG variable instance is both broken and reset.

Figure 3.2: Using a delayed obligation for automatically resetting a BTG variable

```
<Obligation ObligationId="ResetBreakTheGlassObligation" FulfillOn="Permit">
  <AttributeAssignment AttributeId="BTG:(subject-id(S),action-id(A))"/>
   <AttributeAssignment AttributeId="Delay">
         <AttributeValue>30</AttributeValue>
    </AttributeAssignment>
    <AttributeAssignment AttributeId="TimeUnit">
         <AttributeValue>minutes</AttributeValue>|
    </AttributeAssignment>
</Obligation>
```

The ResetBreakTheGlassObligation specifies two optional attribute assignments, called Delay and TimeUnit. Together these make sure that the ResetBreakTheGlassObligation is only executed when a certain time has passed. The value for Delay has to be a positive integer and the supported values for TimeUnit are seconds, minutes, hours and days. This range should be sufficient for most applications. This feature has been implemented using Java's scheduling features. Thus, *no* additional bookkeeping fields are needed in the tables and periodic 'cleaning' of the tables is unnecessary when this feature is used consistently.

When using manual or semi-automatic resetting of BTG-state the policy writer needs to associate both of the reset obligations with the permissions for who is allowed to reset which BTG-variable names. Since we have no control over which subordinate PDP an implementation might use, we cannot specify how its permissions will be encoded. We can however specify which elements the BTG-PDP requires in the request context in order to correctly process the reset obligations that the PDP must return if the reset is to be effective. Consequently we have specified two action identifiers and one action attribute that should be present in the request context. An application is free to insert whatever other elements it wishes into the request context so that the underlying PDP will correctly grant the appropriate users permission to reset the broken glass, and will therefore return the correct obligations to the BTG-PDP.

ResetBreakTheGlassTable is the action that must be specified in the request context in order to reset all the values of a BTG-variable name (or a collection of such names). When permission to execute this action is granted the ResetBreakTheGlassTableObligation must be returned by the underlying PDP. This will have the effect of cleaning the relevant table(s). ResetBreakTheGlass is the action that must be specified in the request context in order to reset a BTG variable instance. When permission to execute this action is granted, the underlying PDP must return the ResetBreakTheGlassObligation. The two optional attribute assignments, Delay and TimeUnit, would normally be absent from the obligation, so that the resetting is immediate. The former obligation doesn't support the Delay and TimeUnit attributes as we do not see the requirement for them.

ResetBreakTheGlassTableObligation does not need to resolve the attributes in the BTG variable name since the table corresponding to the given BTG variable name is completely emptied. (The table does still exist, but immediately after it has been reset its row count is zero.) ResetBreakTheGlassObligation on the other hand does need to resolve the attributes in the BTG variable by using the attribute values in the request context so that the BTG-variable can be resolved into a single row of the table. In order to achieve this we have defined an attribute for the ResetBreakTheGlass action called OriginalRequestContext which should contain a copy of the original request context which was used to break the glass. We use this when resolving the BTG-variable, to ensure that the correct row in the table is reset.

As an example consider a request coming from a Head of Department to reset the BTG-variable BTG:(subject-id,action-id) for the subject 'Hani' and the action 'GET'.

Figure 3.3: An example request context to reset a BTG-variable instance

```
<Request>
 <Subject>
  <Attribute AttributeId="Role">
   <AttributeValue>HeadOfDepartment</AttributeValue>
  </Attribute>
 </Subject>
 <Resource>
   <Attribute AttributeId="resource-id">
    <AttributeValue>PDP specific encoding here</AttributeValue>
   </Attribute>
 </Resource>
 <Action>
   <Attribute AttributeId="action-id">
    <AttributeValue>ResetBreakTheGlass</AttributeValue>
   </Attribute>
   <Attribute AttributeId="OriginalRequestContext">
    <AttributeValue>
     <Request>
       <Subject>
        <Attribute AttributeId="subject-id">
         <AttributeValue>Hani</AttributeValue>
        </Attribute>
       </Subject>
       <Resource>Original resource attributes here</Resource>
       <Action>
        <Attribute AttributeId="action-id">
         <AttributeValue>GET</AttributeValue>
        </Attribute>
       </Action>
       <Environment>Original environment attributes here</Environment>
      </Request>
    </AttributeValue>
   </Attribute>
 </Action>
 <Environment>Any PDP specific data here</Environment>
</Request>
```

We note the following about this request. We don't specify the resource-id as this has to be agreed between the application developer and the policy writer, according to the requirements of the policy language of the underlying PDP. It is only the attachment of the relevant obligations to the appropriate permissions that will actually cause the BTG-state to be reset. The BTG-PDP nor its obligations service care about the resource-id. Apart from the mandatory action-id and the OriginalRequestContext attribute shown in bold in Figure 3.3, the encoding of all other components of the request context is application dependent and must be agreed upon by the application developer and the policy writer of the underlying PDP (including the Subject Role Attribute which is only in Figure 3.3 for illustrative purposes). Thus the policy could for example only allow an administrator to reset the glass between certain hours, and the request context would need to carry the appropriate attributes to allow this PDP permission to be granted. The only requirement the BTG-PDP places on the implementation is that the appropriate action attributes should be present, and the appropriate obligation should be returned along with the grant response.

It should also be noted that the format of the OriginalRequestContext value follows the standard XACML request context schema. This is intentional. Whilst it means that unnecessary elements might be present in a request context, such as the <Resource/> element in the example above, the advantage is that it promotes code reusability as the same code can be used to pick up the BTG-state attribute values as is used elsewhere to manipulate the request context. Also, whilst applications are free to leave out any attributes that they know not to be relevant for resolving the BTG-variable name, nevertheless they can write simpler code which always inserts all the request context values regardless of the underlying permissions.

The reason for having standard action names and a standard attribute are twofold. Just as we don't accept the concept of having to break the glass in order to break the glass we don't accept the concept of having to break the glass in order to reset it. The standard action name helps the BTG-PDP recognize that this request is related to the BTG-protocol and hence when a Deny is returned from the underlying PDP it will not formulate a BTG-request to the underlying PDP. Second, it helps the ObligationsService recognize that the request is BTG-related and hence can treat it in a special way e.g. by using the request context given as the value of the OriginalRequestContext attribute to resolve the BTG-variable name.

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 3.4 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" obligation handling service which sets the BTG state and a "Reset BTG" obligation handling service (see above) which resets the state. These insulate the BTG state layer from the specific implementation details of recording and storing the BTG states. In our initial implementation, these obligation handling services use memory only storage, although they could be replaced by different obligation handling services that use 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.

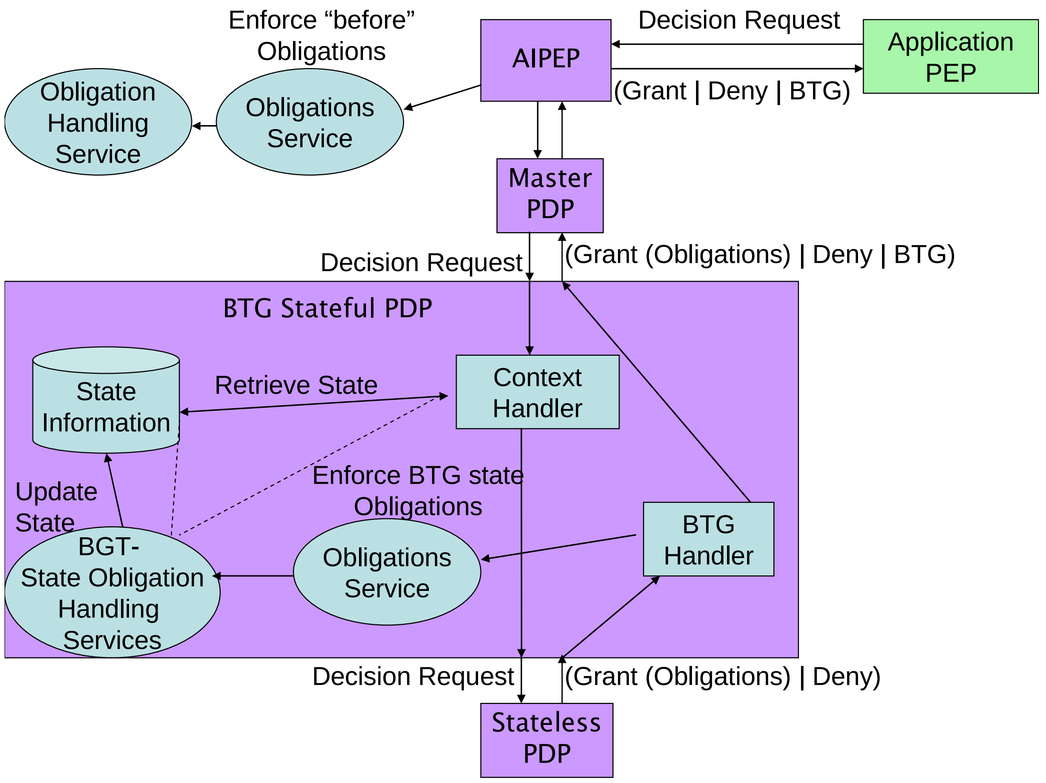


Figure 3.4. The BTG PDP Implementation

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 handling 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)[6] 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 3.4) unaltered. If the Stateless PDP returns

[6] 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.

Deny, then the BTG Handler creates a second request to the Stateless PDP, of the form “Can <the same User> perform operation $O^{BTG(<same\ operation>)}$ 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 $O^{BTG(<same\ operation>)}$ 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 handling 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 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. One 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. An alternative solution would be to create a new Major status code called BTG and to include this with the Deny response. The advantage of this approach is that it is backwards compatible with existing implementations. A PEP which does not support BTG responses from a PDP which does, would treat the BTG response as a Deny and the user would be forbidden access. The PEP could then be enhanced to support the new functionality.
3. The least favourable solution would be for the TAS^3 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 raised this topic with the OASIS XACML TC and the discussions favoured the second approach. A new standardised "BTG" XACML profile was agreed to be written and presented to the XACML TC. Originally it was planned to be a joint effort with Seth Proctor from Sun, but when Sun were taken over by Oracle, Seth left and so could not contribute. Consequently Kent has produced the draft profile itself and this is now under discussion on the XACML TC list.

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 handling 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 to 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.

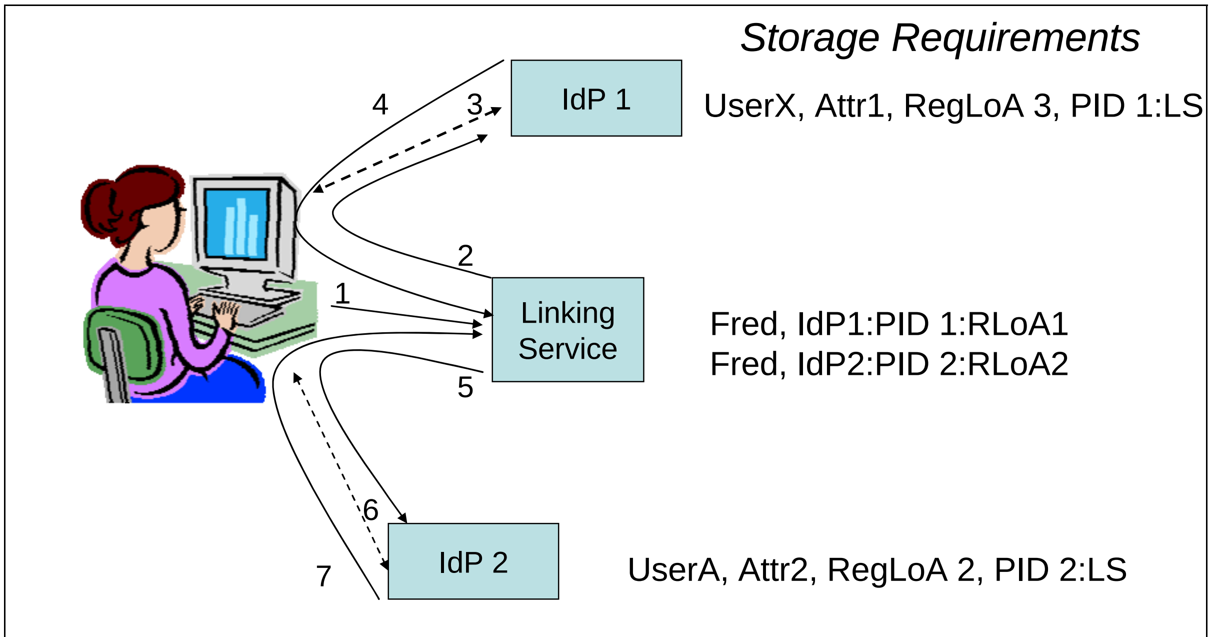
4.1 Conceptual Model

The TAS^3 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 TAS^3 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 (referral) 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 4.1. 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.



1. User contacts her preferred linking service, and chooses one of her IDPs (IDP 1)
2. User is redirected to her chosen IDP
3. User authenticates to IDP, and IDP generates a PID for this user with this linking service (PID 1:LS)
4. User is redirected back to linking service with an authentication assertion that carries the Session LOA and PID. Linking service stores this in the user's linking table entry.
5. User chooses another of her IDPs and the whole process is repeated again

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 4.1.

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 4.1. 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 4.2).

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 4.2. 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 4.2. 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.

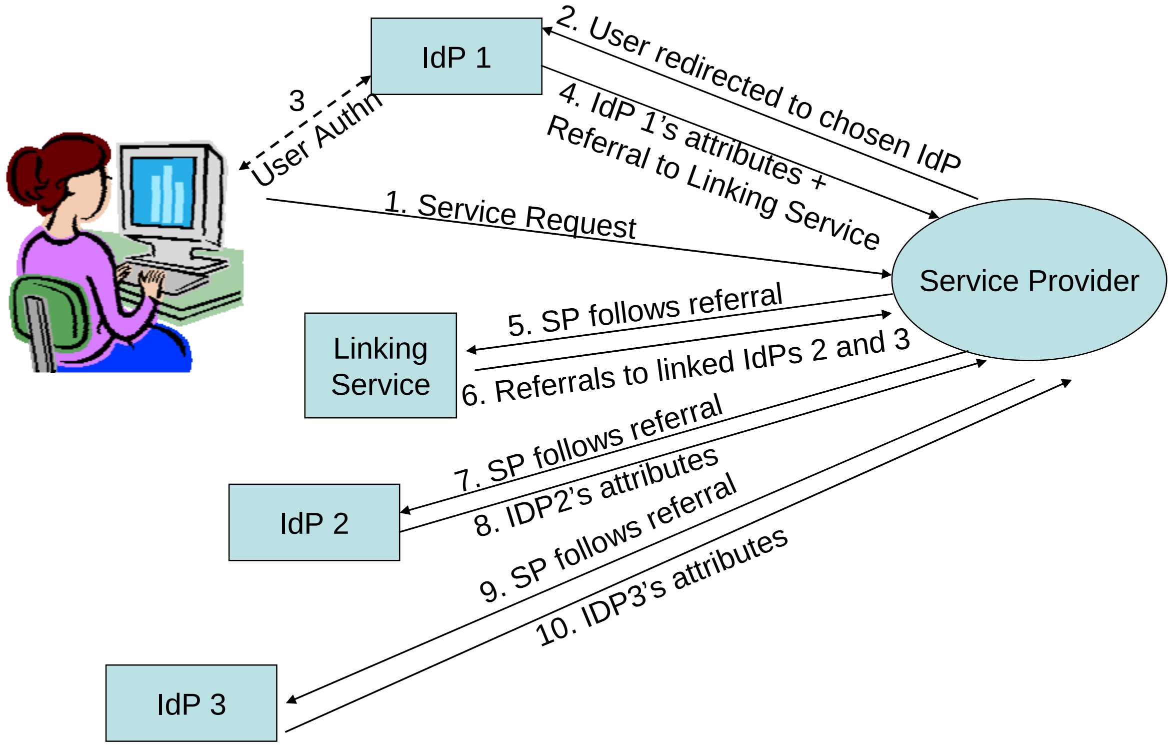


Figure 4.2. Attribute Aggregation by Service Provider

When the linking service receives the referral, it decrypts the permanent identifier and searches for this in its identity provider linking table (see Table 4.1). 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 4.2) 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.

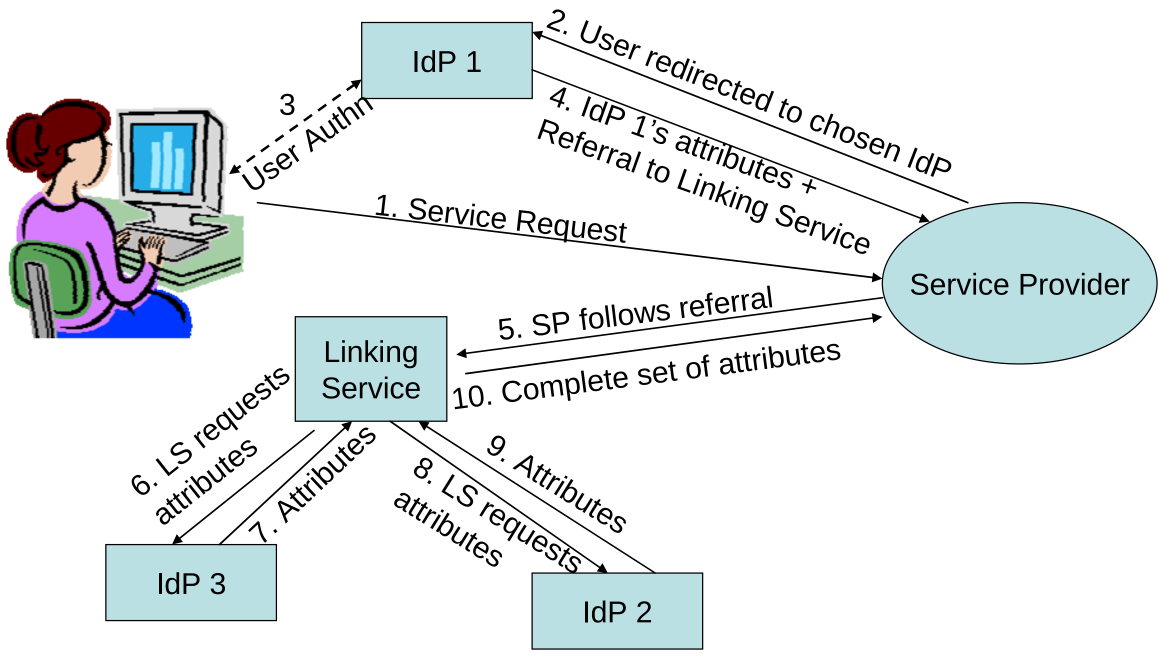

IdP 1
2. User redirected to chosen IdP
3 User Authn
4. IdP 1's attributes + Referral to Linking Service
1. Service Request
Service Provider
5. SP follows referral
Linking Service
10. Complete set of attributes
6. LS requests attributes
7. Attributes
8. LS requests attributes
9. Attributes
IdP 3
IdP 2

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 RequestedAuthnContext element is set to "urn:tas3:referral:store" to indicate that that any persistent identifier that is created should be stored by the system and made available as a referral for subsequent aggregation requests
- 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

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. Referrals are requested by setting the RequestedAuthnContext value of the request to "urn:tas3:referral:release". The SAML response consists of a single SAML assertion which contains: an SSO authentication statement for the requestor, an attribute statement containing the user's attributes and an additional attribute statement that contains a single attribute that holds an additional authentication assertion valid throughout the federation and zero or more referral (EPR) attributes. Once the service provider has received the SAML response it may attempt to access each of the EPR's using the aggregation service protocol mapping described below.

Our original aggregation service 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 implemented this. However, 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. In contrast, 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. Nevertheless we implemented the Discovery Service protocol as recommended by the LA working group, but we found that this mapping was indeed very inefficient and too time consuming for users. Consequently we have now devised a third protocol mapping that only requires one round trip and should perform much better.

This new mapping requires no changes to any existing protocols and combines the functionality of the IDWSF discovery query request with that of the SAML 2.0 Attribute Query message. As the IDWSF discovery request message was primarily used to map a persistent identifier to the transient identifier found in the attribute query message we can replace this message with a single wsse:Security SOAP header element inside the SOAP envelope containing the samlp:AttributeQuery. This header consists of a single SAML 2.0 assertion that is constrained in the following manner:

- The Subject element of the assertion must contain the persistent identifier established between IdP and Linking Service at link registration. This can then be used to identify the internal user that is the Subject of the request.
- A single attribute statement is added containing a single SAML 2.0 attribute. The attribute contains a copy of the signed federation wide authentication assertion issued by the authenticating IdP. This assertion is included as proof of authentication and allows the IdP to check that it trusts the authenticating IdP. This also allows the recipient to ensure that the original act of authentication had a sufficiently high level of assurance (LoA) to release additional attributes. The transient identifier in the Subject element of this assertion should match the Subject element of the attribute query message providing a link between the SOAP header and body.
- The assertion should then be both signed using the creator's private key and encrypted using the public key certificate of the intended recipient.

When an attribute authority (AA) receives a SOAP request created according to this profile it should process the SOAP header prior to processing the body of the message. If the AA is satisfied that the user was authenticated to a sufficiently high level to release additional attributes, a new session is created for the user at the AA. This new session registers the transient identifier (from the federation wide authentication assertion) as belonging to the user identified by the persistent identifier. Once this session has been created the IdP/AA can process the <samlp:AttributeQuery> message found in the body of the SOAP envelope. The <samlp:AttributeQuery> should be constructed in the following manner:

- The Subject of the request must contain the transient identifier present in the federation wide authentication assertion. This provides both a palpable link between message and authentication session and ensures that the SP receives all user attributes under the same name identifier. The ultimate recipient of the attributes must also be specified using the SPNameQualifier to ensure that any subsequently released attributes are encrypted to the correct SP.

- Each attribute required for authorisation may also be requested individually in the message. The AA will then attempt to fulfil each individual attribute subject to any internal attribute release policies. If no attributes are specifically requested then all applicable attributes should be returned to the requestor. The requestor may also signify (to an LS) that it wishes to aggregate additional attributes itself, by including a request for referrals i.e. IDWSF endpoint reference attributes, in the <samlp:AttributeQuery>.

As the transient identifier contained in the Subject of the <samlp:AttributeQuery> message has already been mapped to a new session when processing the header it should then be possible to process the message according to the processing rules described in Section 3.3.2.3 of [11]. Once the message has been processed a new attribute assertion will have been produced; the Subject of this assertion will be the transient identifier contained in the initial request and it should contain each requested attribute available at the IdP/AA. The assertion should then be signed using the IdP/AA's private key and encrypted using the public key of the entity described in the SPNameQualifier attribute of the request's Subject. A <samlp:Response> message containing this assertion is finally returned to the requestor to complete the query.

4.2.2.2 Service Provision using InfoCards/CardSpace Protocols

We also 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.

4.3 User Trials

We conducted a series of user trials using our linking service, and afterwards asked the users a series of questions. The trials were run at the National E-Science Center (NeSC) in Glasgow on the 1st of December 2009. These trials took place using a small group of 9 volunteers using a trial configuration of the linking service software provided by the University of Kent.

The trial was conducted in the following manner. Participants were given a talk demonstrating access to the current DAMES portal at NeSC, which asserts all the roles that a user possesses. The talk then discussed the issues with the existing model, namely that the NeSC identity provider should not be the authoritative source for all the user's attributes, before introducing the linking service. The user was then given a copy of the user instruction document and asked to follow the demonstration detailed in its pages. Once the user had completed the demonstration they were given a questionnaire and asked to return the completed copy via email.

The questionnaire consisted of 19 questions. The first three of these questions attempted to determine the user's understanding of the process and whether the demo was completed successfully or not. Questions 3-6 questions concerned the usability of the linking service. Questions 7 – 10 discussed the user's views on attribute release and which party should control it. Questions 11- 15 dealt with the packaging of the software and where that software should be deployed. Finally questions 16 – 19 acted as a catch all that allowed the user to comment on the software itself and any concerns he/she may have had with it.

When questions needed to elicit a respondent's opinion about a topic, a Likert- type 7-point scale, with answers ranging from 1) Strongly Disagree, 2) disagree, 3) Slightly Disagree 4) Don't Care 5) Slightly Agree 6) Agree to 7) Strongly Agree. Users were also given the option of NA (not applicable) if they did not feel that they could answer the question. The questionnaire was distributed to nine people at the user trial in Stirling and 8 replies were received. A summary of the results is presented below. The full questionnaire can be obtained from http://sec.cs.kent.ac.uk/shintau/user-trial-questions.doc.

4.3.1 User Trial Results

Q1 asked if users understood the basic principles behind the linking service. The majority of respondents strongly agreed (7) with this, with all respondents scoring 5 or above. (One user who had prior knowledge replied 4 since the user documentation did not provide any further knowledge).

The second question asked if the user managed to complete the demonstration successfully. All users did, with the majority replying 7 and all replying 6 or 7.

Q3 asked if the users understood each step of the demo. The results showed that almost all of the respondents understood the purpose of each of the demo's steps with only a single participant having any significant problems.

Q4 asked if the linking service was easy to use. The results showed that all the users agreed that the linking service was quite easy to use with most users (76%) believing that the service was easy to use or very easy to use. This view was reflected in the free form comments in which several users stated that the service was "straightforward". Other comments stated that "the manual is easy to understand and follow, and the interface is very similar to the way we surf the internet." It was also pointed out that whilst the service was easy to use it could easily become very time consuming to perform linking for every SP.

The next question asked the users if they could use the linking service without the help of a manual. The majority 5/8 of the participants agreed with this. Only 1 participant was not sure and 2 were doubtful. This indicates that additional work is still required on the web interface to make sure that the user flow through the software is made clearer. The comments were positive overall but made it clear that in most cases reading the user document at least once would be required in order to use the software fully. Other comments referenced that fact that the demo required the user to remember a large number of URLs such as SP URLs suggesting the need for some form of friendly name for each SP (and IdP). It was also suggested that it would be useful to see the attributes available from each linked IdP to make a more informed decision as to which IdP's to link together.

Q6 asked if the user would be happy to use the Linking Service more generally in taking control over the release of his/her attributes. Whilst 50% believed to some extent that they would be happy to utilise the Linking Service for this purpose the other 50% either had no opinion or slightly disagreed. The comments suggested that whilst participants may be in favour of the idea in theory they would require additional information to be provided on the context of the attributes to be released, before committing to a definite answer. Several users did express a desire for finer control over attribute release but also pointed out that some users would have little interest in such a scheme.

Q7 took a different approach and asked the users if they did not really care which of their attributes were released as long as they got access to the services that they wanted. This question was intended to determine the participant's views on the release of their attributes and the relevant trade off between attribute privacy and security and access to resources. The results were very evenly balanced with roughly one third of the participants agreeing with the question, one third having no strong opinion and the last third disagreeing with the statement. The comments suggest that the participants are aware of the trade off and that their answers are dependent on the types and values of the attributes to be released i.e. they would like greater control over some attributes such as those used for banking but have little interest in those used for less important tasks such as VO participation which they would prefer to leave up to the system to determine.

Q8 asked the users if they could envisage other scenarios where the linking service would be extremely useful. Almost all of the participants could imagine additional scenarios where the linking service could be useful. The comments suggested that whilst users could see the value of the linking service the participants had concerns over the requirements for *all* SP and IdPs to support the linking software.

Q9 asked the users if they preferred all IdPs to give them more control over the release of their attributes. This question was included to determine how the participants felt about the current attribute release mechanisms provided by most IdPs. The results clearly showed that most users did not care (4), believing that the current system is adequate for their purposes. The other participants believed to some degree that greater control should be provided and none of the participants believed that the IdPs should give them less control.

Q10 asked if the users preferred other services or administrators to handle the release of their attributes for them. The group were split down the middle, with a third agreeing, a third disagreeing and third not bothered either way.

Questions 11-15 asked users about the packaging of the software and where the software should be deployed. The answers showed no clear agreement to any of the questions and therefore are not dealt with further here.

Q16 asked in what circumstances would you think the linking software should not be used and/or used with caution? This prompted a variety of responses but with one common theme, namely, the participants believed that the software was still too complicated for non expert users and concerns were raised that they might not be able to configure the software effectively (this refers primarily to the release policy). One user was also concerned that the software should not be used to protect resources of high value (possibly due to the lack of visibility over attribute release or due to the prototype nature of the software). Another user pointed out that the software does not contain any form of redundancy checks on which IdPs are currently available to be queried by the linking service.

Q17 asked which was the best feature of the software. The comments were very positive and suggested that most of the users liked the simplicity and ease of use of the software as a solution to the problem of attribute aggregation. Several users also commented on the potential the software has for accessing several services with single sign on. Comments were also made about the high quality the web interface of the linking service

Q18 asked which was the worst feature of the software. Three of the eight participants commented on the fact that the software brings additional complexity to the federated environment that non expert users may not be willing to undertake. Other comments stated that as the linking service is an untrusted brand, user's have no reason to trust that it is performing its tasks in a privacy preserving manner. There was also a comment on the naming of service providers and identity providers as URLs, which was described as cumbersome.

4.3.2 Conclusions

The results of the user trials made us rethink the user interface design to the linking service. Whilst all the users realised that account linking did provide a useful service (Q8), nevertheless many of the users cared less about linking their attributes and more about gaining access to the service (Q.7,9,10). A fundamental problem is that the users have to link their accounts prior to using any service that requires multiple attributes, and set up their release policies, which may be too complex for many users (Q16). The majority of the users were happy with the solution and found it generally easy to use but the weaknesses were that:

- a greater emphasis should be devoted to making the software more user friendly for "non expert" users who are not familiar with the problem space
- additional information about the attributes provided by each IdP should be made available to allow the user to make more informed consent decisions about the IdPs being linked and the attributes being aggregated.

We therefore decided to redesign the user interface and the protocol mode of interaction, so that the creation of release policies was not necessary and users could dynamically determine which attributes to aggregate together based on the requirements of the service provider at the time of service provision. We decided to base our design on the CardSpace graphical user interface since this is easy for users to comprehend and use. However we need to remove many of the limitations of CardSpace e.g. that only a single card can be chosen. We have called our new design CardSpace in the Cloud, which is described below. This is fundamentally different to our original CardSpace protocol mapping described in 4.2.2.2, as the latter still only allows the user to select a single card and the aggregation is provided by the linking service in the same way as in these user trials. Essentially the new design integrates an enhanced CardSpace GUI and identity selector onto a backend linking service.

4.4 CardSpace in the Cloud

4.4.1 Introduction

Information cards are a core component of Microsoft's Cardspace identity management and authorisation system. From a usability perspective InfoCards provide a metaphor that is familiar to users – that of plastic cards in a wallet. The simple clicking of a card simultaneously provides user consent and submission of personal attributes to a service provider (SP). Cards may be self issued or managed, meaning that the attributes originate from either the user herself or an Identity Provider (IdP). From a security perspective they significantly reduce the risk from phishing attacks, they provide privacy protection of the user's personal information, and good assurance to the service provider (SP) that the user has the managed attributes that are being claimed. Unfortunately it has failed to be widely adopted, probably because the model suffers from a number of significant disadvantages. CardSpace only supports a limited number of authentication mechanisms with an IdP (un/pw, X.509 certificate and Kerberos V5) and adding new mechanisms such as voice biometrics or one-time passwords is impossible without significant changes to the protocol flows. The user's cards are held in a fat client on the desktop (the Identity Selector) which constrains mobility and limits the choice of end user devices. Furthermore, Cardspace only allows a single multi-attribute card to be used in each transaction, which is a serious limitation. In the physical world users typically have lots of plastic cards issued by many different IdPs. Each card typically holds only one (or very few closely related) user attribute(s), along with its validity period, a user identifier, a mechanism to authenticate the user (usually a signature or PIN, but could be a photograph as well), and details of the issuer. (Other contents such as holograms and chips are there to ensure the authenticity of the physical card and the attribute assertion (or claim) that it makes. They do not provide additional attributes of the user.) Users control their own privacy by determining which cards to release to whom. In the InfoCard model this means that users should be able to select and use multiple cards in a single transaction, with each card revealing just one (or a few closely related) attributes. This mechanism then becomes an attractive alternative to UProve and Idemix anonymous credentials, since it uses existing ubiquitous asymmetric cryptography.

By way of a motivating example, say a user wishes to make an anonymous online hotel booking, in which she needs to prove she has a valid credit card to pay, and use her frequent flyer card to score air miles, whilst providing a (self-asserted) alias name and address. Furthermore, she does not want to have to use multiple usernames and passwords in order to send these multiple cards. Because she is sending her credit card details, she wants to use her credit card IdP, which uses strong authentication, to authenticate her. Furthermore it does not reveal her credit card details to the hotel, but instead sends it a signed assertion with a session identifier confirming that the user holds a valid credit card, and that payments can be charged to the session id.

4.4.2 Conceptual Model

Our model still assumes that the user is the only person who knows about all of her managed cards (i.e. IdP accounts), and that she does not wish any IdP to know about all of her other accounts. This mirrors real life today. We further assume that all IdPs and SPs are part of an Internet wide federation with pre-existing independently determined trust relationships between themselves (as in the global world of credit cards today). We introduce a new SP into the federation, called a Linking Identity Selector (LIS). The LIS is a combination of the original linking service described above and an enhanced CardSpace Identity Selector. The LIS holds links to the user's various IdP accounts, and allows the user to select multiple cards (IdPs) at service provision time. Importantly, the LIS helps to protect the privacy of the user since each IdP links to the LIS instead of to each other. Furthermore the LIS does not have any knowledge of who the user actually is or what attributes are held by each individual IdP, except for those attribute types (but not values) that the user chooses to release to the LIS when she links her accounts together. Linking simply requires the LIS and IdP to share a secret persistent ID (PId) between themselves for the user. The user is free to choose any LIS in the federation and is not bound to any single provider. The user may have different accounts at multiple LISs if desired. Each LIS is trusted by (a subset of) IdPs and SPs to hold the user's PIds confidentially and securely and to only release these details back to their respective IdPs when requested by the user. The scheme has two phases: registration, and service provision.

4.4.2.1 IdP/Card Registration

We introduce a pared down InfoCard schema that contains no personal data, only the picture/logo and name of the IdP along with the metadata needed to access it. The advantage of this design is that a single InfoCard is valid for all users, and cards can be made publically available. If they are intercepted or lost they don't affect the user's privacy (unlike current InfoCards). We further propose that each IdP hosts an InfoCard well known address e.g. http://idp.com/InfoCard/ where users can obtain cards. This removes a current limitation of today's managed cards, i.e. no-one knows how or where to get them. When a user wishes to retrieve an InfoCard in order to register at a LIS, she can navigate to this well known address using any browser and download it to her PC (as a file).

Since our LIS runs in the cloud, there is no reason for users to need to import and export their cards as they move between devices, as with current InfoCards, since all their devices can contact the LIS in the cloud. Users access the LIS with a normal web browser and a simple new plug-in module, which is advantageous to using the thick client of current CardSpace systems. The user may upload one of her InfoCard files to the LIS, or the LIS may already hold cards for all the IdPs in the federation it has trust relationships with. Once the LIS knows which card the user wishes to use, it will redirect the user to the authentication service of that IdP, using the metadata on the card. This will prompt the user to login/authenticate using any mechanism the IdP has chosen. We thus devolve the act of authentication from the identity selector, thereby freeing the IdPs to use any authentication mechanisms they wish. We use the standard SAMLv2 protocol for this protocol exhcange, with the LIS acting as a SP.

The LIS requests an authentication and attribute type assertion from the IdP. The IdP returns a PId which is used as a pair-wise secret between the LIS and the IdP, to identify the user's account at both ends in all future communications. After authenticating the user, the IdP asks the user which attribute type(s) she wants to include in her linked card at the LIS, and includes this in the response. The NIST Level of Assurance (LoA) is also included in the response exactly as described previously in section 4.1.2. When the LIS receives an unknown PId, it creates a new account for the (unknown) user in its internal database and stores the returned attribute type(s) and LoA in the user's card for this IdP. (The attribute type is used at service provision time to determine if the IdP's card is selectable or not.) When the LIS receives an existing PId from the IdP, it locates the (unknown) user's existing account in its database. If the user wishes to link additional IdP accounts to this LIS account then she submits or chooses another IdP card and is redirected to that IdP. The LIS requests another PId and set of attribute types from this IdP and adds these to the same LIS account. The LIS therefore holds a set of secret PIds for each user, along with their associated LoAs and attribute types.

4.4.2.2 Service Provision

We introduce a thin client module that is pluggable into the user's web browser. This module is used to discover the user's LIS, to redirect the user there, and finally to return the claims to the

SP as in the existing CardSpace system. The only difference with current InfoCard systems is that multiple cryptographically linked claims (attributes) from multiple IdPs may now be provided to the SP, as determined by the user.

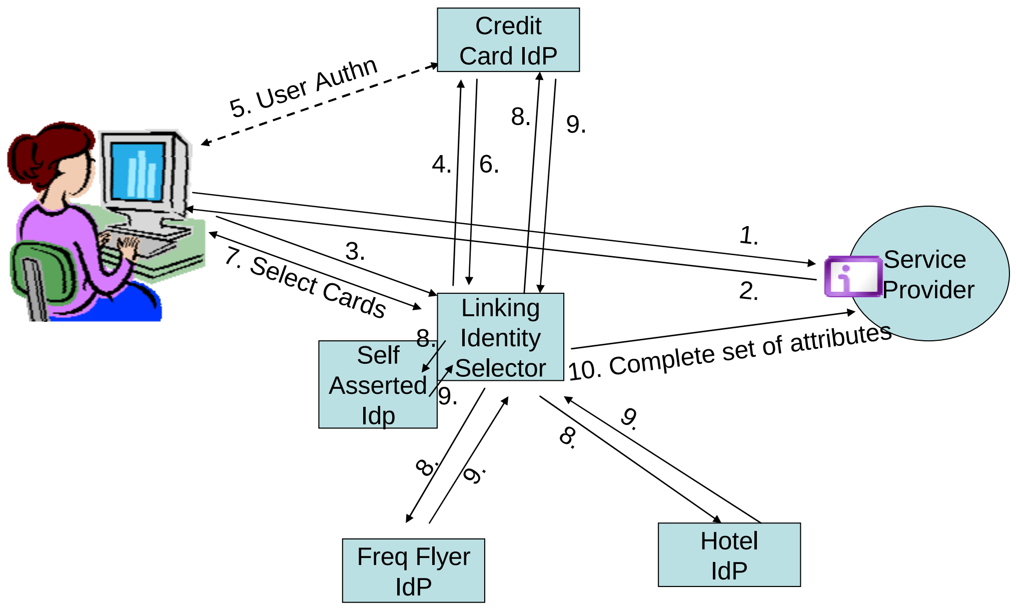
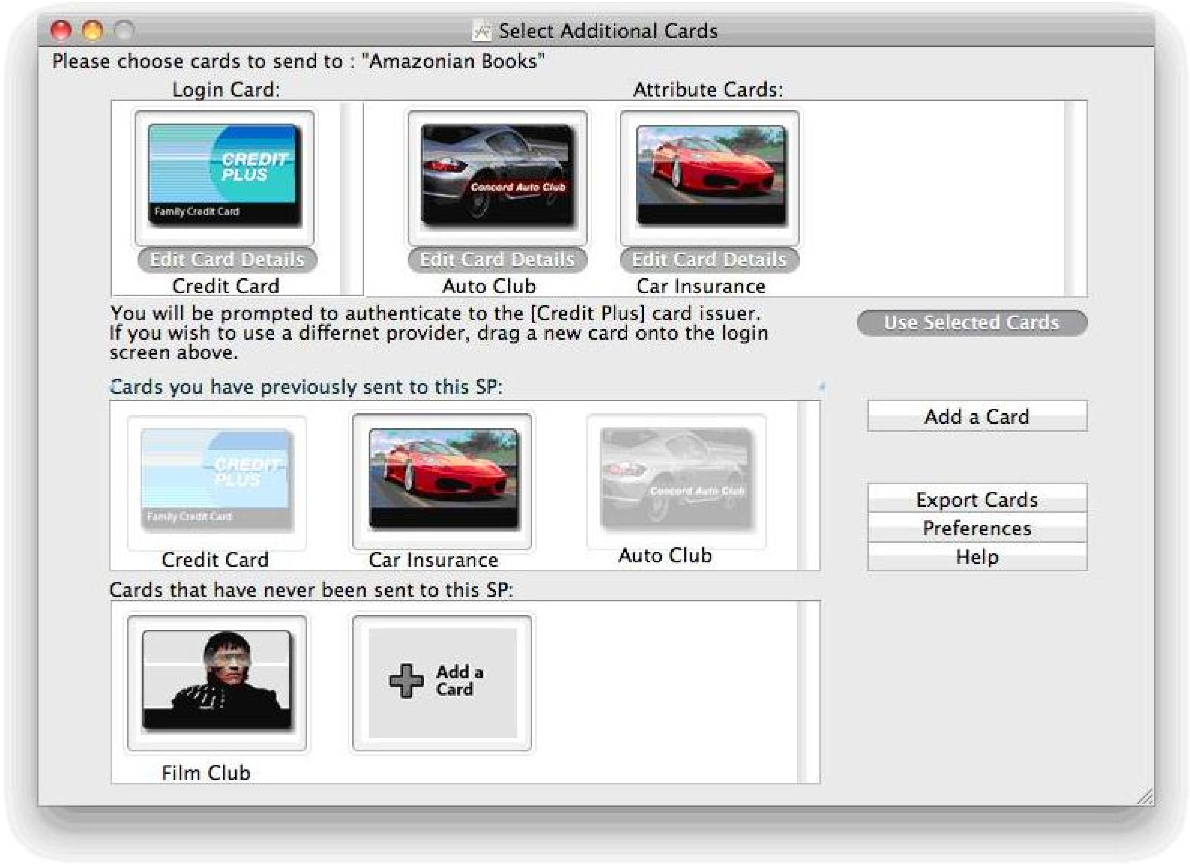


Figure 4.4 Service Provision Protocol Interactions with CardSpace in Cloud

The user navigates to an InfoCard enabled HTML page at the SP's site and clicks on the InfoCard icon using their browser (step 1). The returned page (step 2) contains an embedded InfoCard MIME object that causes the browser to invoke our new plug-in module. The module downloads the SP's security policy. We have defined an enhanced security policy that allows multiple IdPs issuing different attributes to be specified. This is a significant enhancement to the CardSpace framework. The module next discovers the location of the user's LIS in one of two ways. The user may have previously bookmarked the home page of her LIS in a reserved bookmarks folder that can be accessed by the module. The module can then display this/these bookmarks to the user and let her choose her preferred LIS. Alternatively the user can directly enter a URL into the module. This allows our system to be used on any Internet café computer without releasing personal information to other users. Our system prevents phishing attacks since a fraudulent SP is not able to redirect the user to a LIS of its own choosing.

When the user has selected her LIS, the module establishes a TLS connection with it to protect all future communications (step 3). The module forwards the SP's policy to the LIS and asks that the attributes be returned encrypted for the SP. When the LIS receives this message, if this is the first interaction with the user/ browser (i.e. no cookie returned) it acts as a Where Are

You From (WAYF) service and displays a page showing the user a list of all the IdPs that it has trust relationships with. The user chooses one, is redirected there (step 4), and invited to authenticate using the IdP's supported mechanism (step 5). The IdP responds to the LIS by returning a SAML authentication assertion containing the LoA, a random session ID and a "referral" attribute which contains the user's PId encrypted for the LIS (step 6). The LIS is able to decrypt the PId and access the user's account. It displays all the user's linked cards in an Identity Selector like page (step 7). If a cookie was provided by the browser module, the LIS decrypts it, and opens the user's account immediately (without IdP authentication), and displays all her cards in step 7.



Please choose cards to send to : "Amazonian Books"

You will be prompted to authenticate to the [Credit Plus] card issuer.
If you wish to use a differnet provider, drag a new card onto the login screen above.

Figure 4.5 The Linking Identity Selector GUI

The Identity Selector page comprises three windows (see Figure 4.5). The bottom two windows are similar to today's Identity Selector. The top window contains cards that have already been selected, the middle window shows cards that have previously been sent to this SP (but not yet selected), whilst the bottom window shows cards that have never been sent to this SP (which tells the user to exercise extra care when choosing them). Those cards that match the SP's requirements are lit up and those that don't are greyed out (and not selectable). The top window initially only contains either one card, that of the authenticating IdP, or no card (if a cookie was used). As the user clicks on cards in either of the two lower windows two things happen. Firstly the card is removed from the lower window and displayed in the top window. Secondly any cards which are no longer needed to fulfil the SP's policy are greyed out in the lower windows. For example, if the SP's policy says that it needs a credit card from either Visa, Mastercard or Amex, and the user clicks on her Visa card, her other credit card icons will be greyed out in the lower windows. Once the combined set of selected cards match the set of attribute types requested by the SP, the "Use Selected Cards" button is enabled. (The LIS remembers this selection so that they can appear in the middle window the next time the same SP is contacted.) Because the LIS is in the cloud and not on the desktop, when the user moves from device to device she will not keep being told she has not sent cards previously to an SP when in fact she has, as in the current CardSpace system.

If the user has not yet authenticated because a cookie was used, the user is now redirected to the IdP of the first card chosen (step 4) and authentication is performed as before. Finally the LIS queries each of the chosen IdPs for the user's attributes (step 8.). All the chosen IdPs, including the self asserted IdP and the authenticating IdP, are contacted in the same way. The query comprises: a SAML attribute query requesting the chosen subset of attributes required by the SP, a referral containing a PId encrypted to the recipient IdP, which points to the user's account at the IdP, and the original authentication token containing the random session ID and LoA. The recipient IdP uses the latter to determine whether it trusts the initial act of authentication by the authenticating IdP. If it does not, the recipient IdP returns an error to the LIS. If it does trust the authenticating IdP then it generates a claim containing the user's attributes encrypted to the SP (step 9). The user is identified in this assertion by the random session identifier contained in the authentication token. The LIS stores the returned claim until all the queried IdPs have replied.

Once all the attribute assertions have been collected by the LIS it generates a response to the original request made by the browser module. This response contains the authentication assertion and each of the encrypted attribute claims returned from the IdPs. When the browser receives the response it forwards it to the SP (step 10). The SP receives a single authentication token and multiple attribute claims from multiple IdPs which all contain the same random session identifier. Since the SP trusts all the IdPs as being authoritative for their attributes, it can be assured that the same user possesses all of the returned attributes, and has been successfully authenticated to a particular level of assurance.

4.4.2.3 Next Steps

We are currently implementing the CardSpace in Cloud software, and once this is finished we will carry out user trials and report on the results at the end of the project.

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 controller 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.

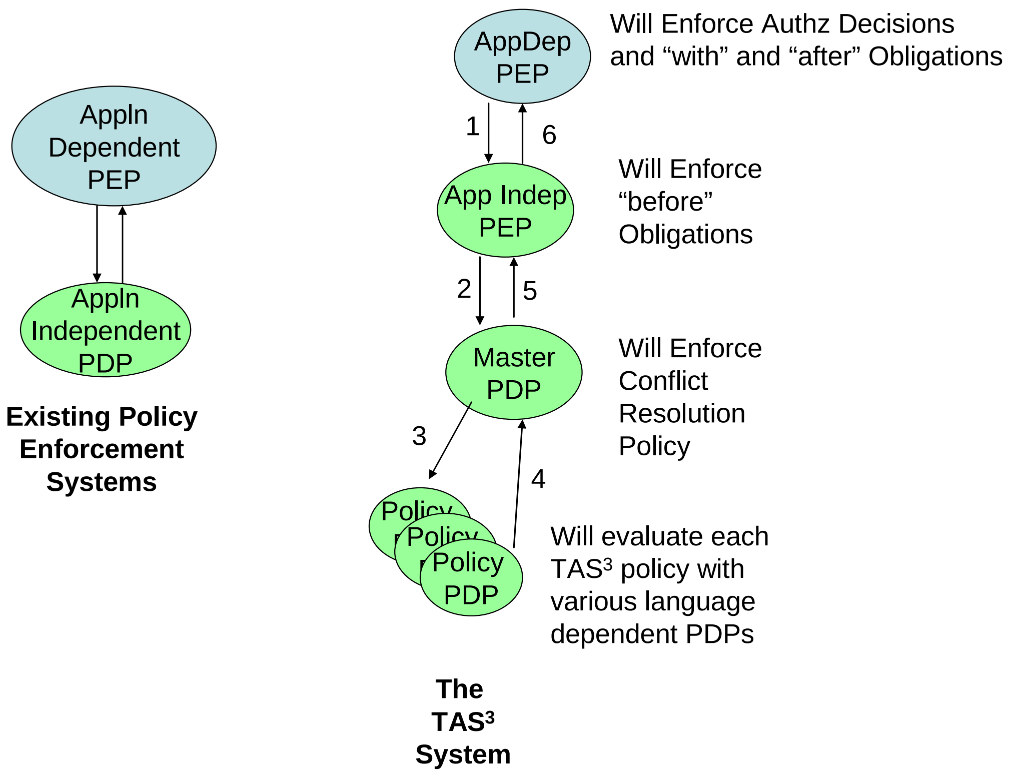

AppDep PEP
Will Enforce Authz Decisions and "with" and "after" Obligations
1
6
App Indep PEP
Will Enforce "before" Obligations
2
5
Master PDP
Will Enforce Conflict Resolution Policy
3
4
Policy PDP
Will evaluate each TAS³ policy with various language dependent PDPs
The TAS³ System
Appln Dependent PEP
Appln Independent PDP
Existing Policy Enforcement Systems

Figure 5.1. The TAS³ PEP-PDP Infrastructure

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 for each authorization decision request. Each of the policy PDPs supports the same interface, which is either the pure XACML request-response context [15] or the latter embedded in a SAML assertion [34]. 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 our design isolates 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 policy type(s) and policy language(s) that each subordinate PDP supports, and therefore when it is passed a sticky policy (see figure 5.2 part 2) it knows which PDP to give each component of the sticky policy to.

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
   xmlns="tas3:to:be:decided:namespace"
   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 local file store as there seem to be 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.
    Version 7. 2 July 2010.
    The DataResource can be any data or resource which requires a sticky policy.
    Resource Types holds the type(s) of resource that are contained
    in the DataResource e.g. it could be a computer system or an email message
    or some PII.
    Any number of policies can be stuck to a DataResource.
    The XML signature is optional because applications may choose to
    secure the PAD using alternate means, e.g. SSL/TLS.
      </xs:documentation>
   </xs:annotation>
   <xs:sequence>
      <xs:element ref="DataResource"/>
      <xs:element name="DataResourceTypes" type="ResourceTypes"/>
      <xs:element ref="StickyPolicy" maxOccurs="unbounded"/>
      <xs:element ref="ds:Signature" minOccurs="0"/>
   </xs:sequence>
</xs:complexType>

<xs:complexType name="ResourceTypes">
  <xs:sequence>
    <xs:element name="ResourceType" type="xs:anyURI" maxOccurs="unbounded"/>
  </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 Policy ID specifies the globally unique ID of the policy.
        The PolicyLanguage specifies the language the policy is written in.
        The PolicyAuthor specifies the person who wrote the policy.
        Time of Creation specifies when the policy was written.
        Expiry time specifies after what time the policy should be ignored. Infinity is the default.
        Resource Type specifies the type(s) of resource this policy refers to.
        The PolicyContents contains the policy written in the language
        specified in PolicyLanguage.
        The PolicyType specifies what type of policy this is.
      </xs:documentation>
   </xs:annotation>
   <xs:sequence>
      <xs:element name="PolicyAuthor" type="saml:NameIDType"/>
      <xs:element name="PolicyResourceTypes" type="ResourceTypes" />
      <xs:element ref="PolicyContents"/>
   </xs:sequence>
   <xs:attribute name="PolicyID" type="xs:anyURI" use="required"/>
   <xs:attribute name="PolicyLanguage" type="xs:anyURI" use="required"/>
   <xs:attribute name="PolicyType" type="xs:anyURI" use="required"/>
   <xs:attribute name="TimeOfCreation" type="xs:dateTime" use="required"/>
   <xs:attribute name="ExpiryTime" type="xs:dateTime" use="optional"/>
 </xs:complexType>
<xs:element name="PolicyContents" type="AnyXMLType"/>
<xs:element name="DataResource" type="AnyXMLType"/>
 <xs:complexType name="AnyXMLType" mixed="true">
   <xs:sequence>
      <xs:any minOccurs="0" maxOccurs="unbounded" namespace="##any"
 processContents="lax">
         <xs:annotation>
            <xs:documentation>
               Any xml content is allowed in this element.
            </xs:documentation>
         </xs:annotation>
      </xs:any>
   </xs:sequence>
</xs:complexType>
</xs:schema>
```

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

When components such as the AIPEP and PEP, or Master PDP and subordinate PDP 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][7].

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 design is that any number of conflict resolution rules can be configured into the Master PDP. Each policy rule will match a specific request context (or set of request contexts), and when a rule matches it will return an obligation containing the conflict resolution algorithm that is to be used for this request context when querying the subordinate PDPs. If no rule matches a given request context, then a default conflict resolution algorithm will be returned. In the current design the conflict resolution policy has to be set by the data controller of each resource site, and we will provide an exemplar policy based on EC data protection legislation. The provision of dynamic conflict resolution policies is for further study.

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

[7] This protocol has a field for an XACML policy or policy reference to be inserted, but this does not allow a TAS^3 sticky policy in any non-XACML language to be passed along with the request-response context. We have proposed a minor enhancement to the SAML-XACML protocol to allow it to carry a policy in any language or languages that the PDP supports. The XACML TC has agreed to such an enhancement but at the time of writing we are waiting for a new release of the OASIS specification to include this. Consequently in our first implementation we have used the SAML <Extensions> field to carry the StickyPolicy construct (see Appendix 4). This will be changed once we have an official standard way of doing this.

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 Appendix 4 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 the Conflict Resolution Policy (CRP) which covers access to its (static) resources. In future it may be possible to dynamically present a Conflict Resolution Policy taken from a sticky policy attached to dynamic data, and passed as part of the authorisation decision request [25, 34]. Who should be the author of a conflict resolution policy? For static resources it is clearly the resource owner. For dynamic PII resources the answer is more difficult. The solution we are proposing to adopt is that it is the law which sets the conflict resolution policy, and we will provide an exemplar for organization to use. Where the law does not have a policy, then the data controller can set it.

A conflict resolution policy (CRP) consists of multiple conflict resolution rules (CRRs). The default CRP is read in at program initialisation time and in future additional CRRs may be dynamically obtained from the subjects' and issuers' sticky policies. Each conflict resolution rule (CRR) comprises:

- a condition, which is tested against the request context by the Master PDP, to see if the attached decision combining rule should be used,
- a decision combining rule (DCR),
- optionally an ordering of policy authors (to be used by FirstApplicable DCR)
- the CRR author and
- the time of CRR creation.

A DCR can take one of five values: FirstApplicable, DenyOverrides, GrantOverrides, SpecificOverrides or MajorityWins which applies to the decisions returned by the subordinate PDPs. The DCRs will be discussed shortly.

The Master PDP is called by the AIPEP and is passed the list of PDPs to call and the request context. From the request context it will get the information such as requester, requested resource type, issuer and data subject of the requested resource. The Master PDP has all the CRRs defined by different authors as well as a default one. From the request context it knows the issuer and data subjects and so can determine the relevant CRRs. It will order the CRRs of law, issuer, data subject and keeper sequentially. For the same author the CRRs will be ordered according to the time of creation. For example the CRRs for a data subject could be

CRR1= if (resourceType=PII, requester=myfriend, requestDate > 10.12.2010) DCR= DenyOverride

CRR2=if (resourceType=PII, requester=myemployer) DCR=GrantOverride

CRR3=if (resourceType=PII) DCR=MajorityWins

All the conditions of a CRR need to match with the request context for it to be applicable. The CRR from the ordered CRR queue will be tested one by one against the request context. If the CRR conditions match the request context the CRR is chosen. If the CRR conditions do not match the request context the next CRR from the queue will be tested. The default CRR (which has DCR=DenyOverrides) will be placed at the end of CRR queue and it will only be reached when no other CRR conditions match the request context. The PDPs are called according to the DCR of the chosen CRR.

Each PDP can return 5 different results –Grant, Deny, BTG, NotApplicable and Indeterminate. NotApplicable means that the PDP has no policy covering the authorisation request. Indeterminate means that the request context is either mal-formed e.g. a String value is found in place of an Integer, or is missing some vital information so that the PDP does not currently know the answer.

BTG (Break the Glass) [58] means that the requestor is currently not allowed access but can break the glass to gain access to the resource if he so wishes. In this case his activity will be monitored and he will be made accountable for his actions. BTG provides a facility for emergency access.

If DCR=FirstApplicable the CRR is accompanied by a precedence rule (OrderOfAuthors) which says the order in which to call the PDPs. For example, if (resourceType=PII, requestor=data subject) DCR=FirstApplicable, OrderOfAuthor=law, dataSubject, keeper. The Master PDP calls each subordinate PDP in order (according to the order of authors), and stops processing when the first Grant or Deny decision is obtained.

For SpecificOverrides the decision returned by the PDP containing a rule with a more specific subject/ resource has priority over the PDP with a rule containing less specific subject/resource. As the master PDP does not have the rule the PDP needs to return the rule together with the decision in order to determine which PDP has the most specific subject/resource. The Master PDP will call the Ontology Mapping Server OBIS (see section 10) to determine which of the returned rules has the most specific subject first. If multiple PDP rules have the most specific subject the Master PDP will call the Ontology Mapping Server again to find the most specific resource among the rules having the most specific subjects. If multiple

PDP rules have the same most specific subject and resource the decision of PDP with the latest creation time will be chosen. If any of the PDPs do not return a rule but a decision only then it is not possible to implement SpecificOverrides as there is no way to determine whether the non rule returning PDP had the most specific subject or not. In that case a default rule (Deny Override) will be implemented as a fallback strategy.

For DenyOverrides and GrantOverrides the Master PDP will call all the subordinate PDPs and will combine the decisions using the following semantics:

- DenyOverrides – A Deny result overrides all other results. The precedence of results for deny override is Deny>Indeterminate>BTG>Grant>NotApplicable.
- GrantOverrides – A Grant result overrides all other results. The precedence of results for grant override is Grant>BTG>Indeterminate>Deny>NotApplicable

When a final result returned by the Master PDP is Grant (or Deny) the obligations of all the PDPs returning a Grant (or Deny) result are merged to form the final obligation.

For MajorityWins all the PDPs will be called and the final decision (Grant/Deny) will depend on the returned decision of the majority number of PDPs. If the same numbers of PDP return Grant and Deny then BTG will count as a Grant, but if there is no BTG then Deny will be the final answer. If none of the PDP return Grant/Deny then BTG overrides Indeterminate which overrides NotApplicable.

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 TAS^3 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.

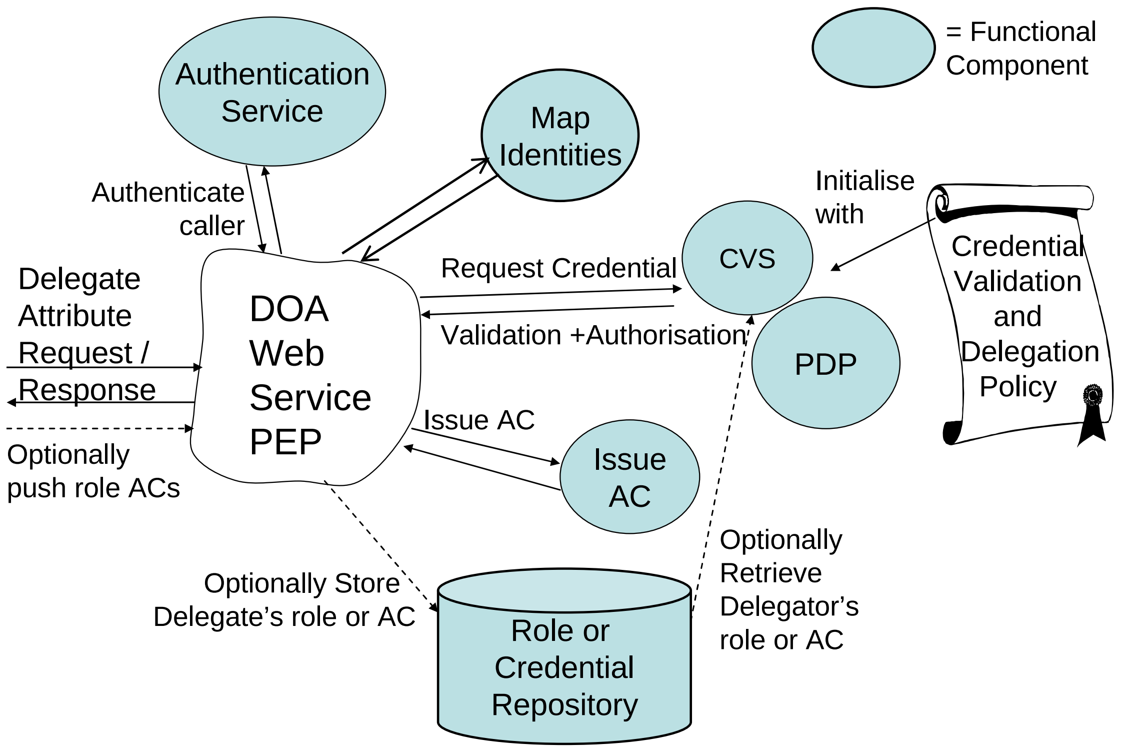

= Functional Component
Authentication Service
Map Identities
Authenticate caller
Initialise with
CVS
Credential Validation and Delegation Policy
Request Credential
Delegate Attribute Request / Response
DOA Web Service PEP
Validation +Authorisation
PDP
Optionally push role ACs
Issue AC
Issue AC
Optionally Store Delegate's role or AC
Role or Credential Repository
Optionally Retrieve Delegator's role or AC

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[8] or when the authentication mechanism uses a different name form to that stored in the issued credentials[9]. 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 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

[8] 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.

[9] 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.

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 later in this section.

6.3 Implementing the Delegation of Authority Web Service

The University of Kent already had a Delegation Issuing Service (DIS), described in [16], and this has been used as the background technology to be enhanced for TAS^3. The DIS originally supported the issuing of delegated credentials as signed X.509 attribute certificates (ACs) which may be long or short lived, and may be revocable. The first task was to add the issuing of signed SAML assertions which are short lived and non-revocable, whilst we still only store long lived ACs in the attached credential repository, which is 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 one or more short lived SAML assertions be created based on his long lived stored credential.

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 URL (step 2 in Figure 6.2) the delegator will pass the secret to the delegate (step 3). The DIS needs supplementing with a client facing service, called the DIS-SP in figure 6.2, which is the service designed to accept secret URLs from delegates and ensure that they are properly authenticated and identified.

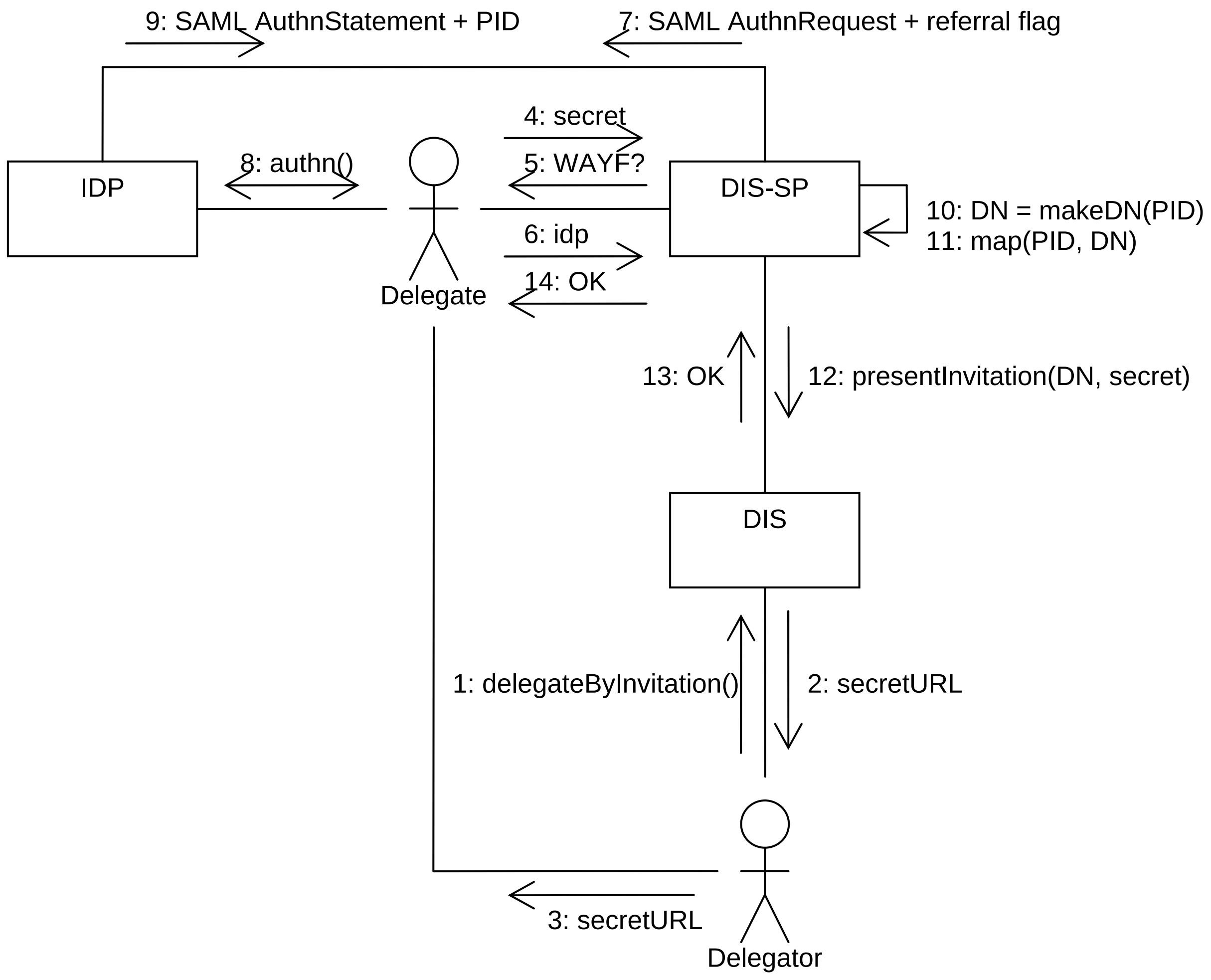


Figure 6.2 Delegation by Invitation

The delegate will click on the received URL and be presented with a Where Are You From (WAYF) screen by the DIS-SP, which re-directs the user to his chosen IdP (steps 5 and 6). The DIS-SP requests the IdP to return a SAML authentication assertion for the delegate containing a persistent ID (PID), and to use this persistent ID in subsequent returned referrals as described in section 4 (step 7). The IdP asks the user to authenticate (step 8) and returns an authentication assertion to the DIS-SP (step 9). The DIS-SP converts the PID into an LDAP DN which will be used to identify the user in all subsequent interactions with the DIS (step 10) and the DIS-SP stores this mapping (step 11). The DIS-SP contacts the DIS, presenting the secret and the new DN for the delegate (step 12). The DIS will check if an entry exists with the secret (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 be delegated the AC. If so, it will issue the AC using the DN of the delegate provided by the DIS-SP and store this in LDAP. If not it will reject the request. 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 have been defined: "action" and "target". Values for 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 any delegate is allowed these attributes (this allows for downgrading of the roles). Finally the DIS will make a call to the PDP passing the (possibly 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 any delegate and that the delegate is allowed to perform this delegated action on this target. The DIS will then return the secret URL to the delegator and store an AC for the secret DN containing the specified action and target. The procedure then continues as before.

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.

6.3.1 Using the Delegated Credential

In order to support the issuing of multiple short lived SAML assertions from a single act of delegation, a new component, the DIS-AA is introduced. This acts as a SAML attribute authority and returns SAML attribute assertions to SPs who can prove that the delegate has recently been authenticated to them. It works as follows. The delegator contacts the SP to request a service (step 1 of Figure 6.3). The delegate is redirected to his chosen IdP (steps 2 to 3), and the SP asks the IdP for an authentication assertion and referrals to other AAs or Linking Services (step 4) that may hold additional attributes for this user. The delegate authenticates (step 5) to the IdP and it returns an authentication assertion and a referral[10] (to the DIS-AA) to the SP (step 6).

[10] In fact the SP will return all the referrals that it has, which may be to one or more linking services as well as the current DIS and other DISs.

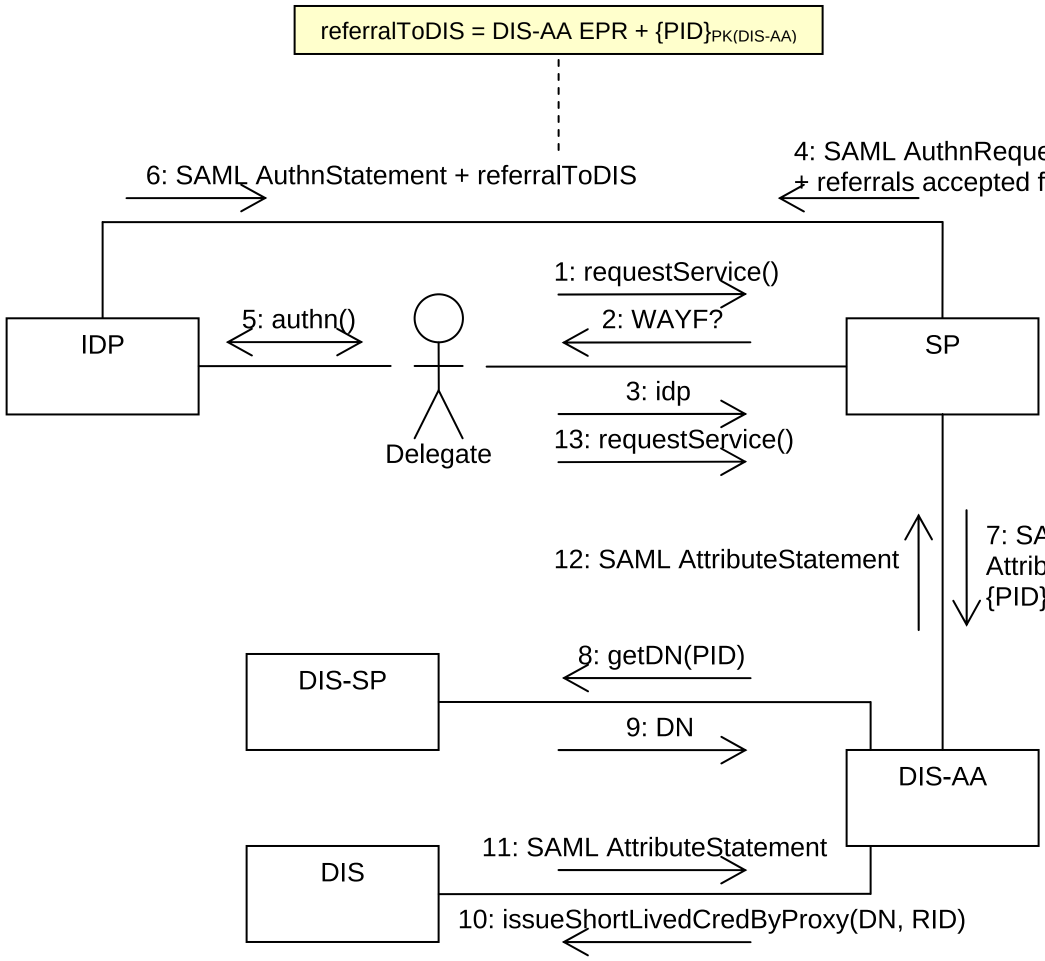

referralToDIS = DIS-AA EPR + {PID}PK(DIS-AA)
6: SAML AuthnStatement + referralToDIS
4: SAML AuthnRequest + referrals accepted flag
1: requestService()
2: WAYF?
5: authn()
IDP
SP
Delegate
3: idp
13: requestService()
12: SAML AttributeStatement
7: SAML AttributeQuery + {PID}PK(DIS-AA)
8: getDN(PID)
DIS-SP
9: DN
DIS-AA
11: SAML AttributeStatement
DIS
10: issueShortLivedCredByProxy(DN, RID)

Figure 6.3 SP Obtaining delegated SAML Assertion

The SP acts on the referral by contacting the DIS-AA and asking for any attributes it has. The DIS-AA decrypts the user's PID in the referral, then asks the DIS-SP to return the DN of the user identified by the PID (steps 8 and 9). Finally the DIS-AA asks the DIS to issue a short lived SAML assertion for the user identified by the DN, but to assign it to the random ID (RID) provided by the IdP in the authentication assertion (step 10). 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 SAML credential which is a subset of the long lived stored ones. This is returned to the DIS-AA (step 11) which returns it to the SP (step 12).

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[11]. 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

[11] XACML also supports other answers: indeterminate (meaning an error) and not applicable (meaning no applicable policy), but these are conceptually other forms of No.

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)[12]. 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 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

[12] 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.

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).

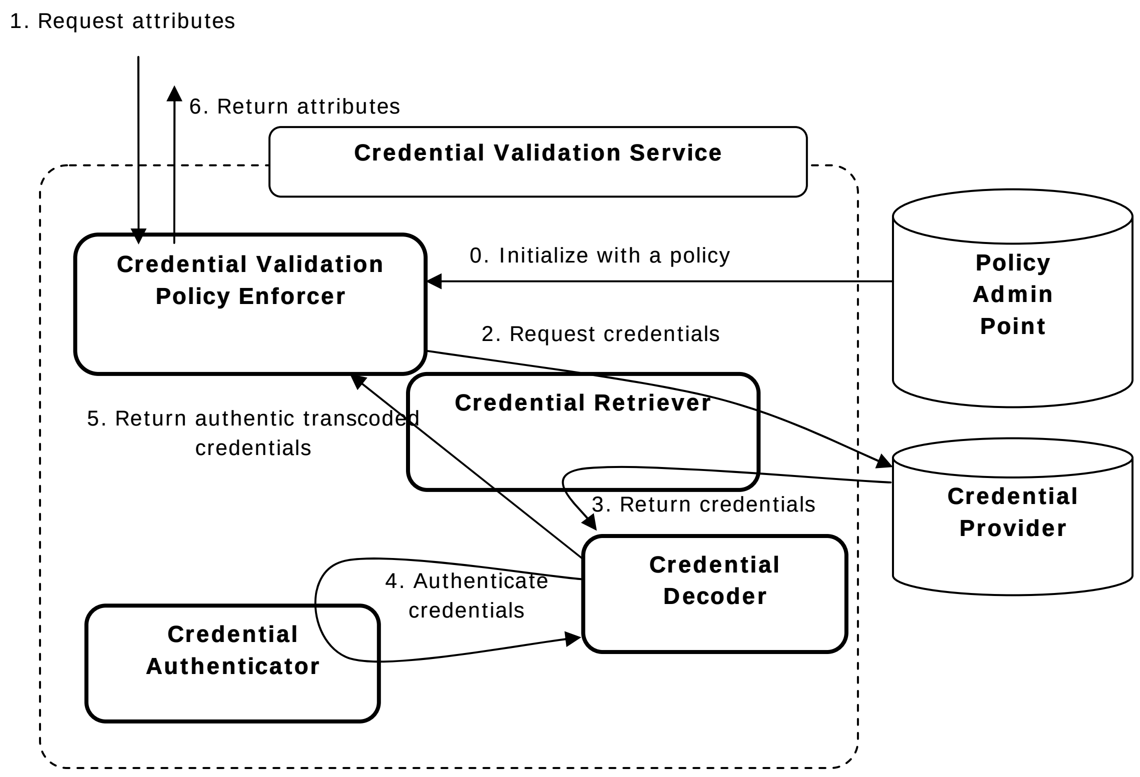


Figure 6.2. Data Flow Diagram for Credential Validation Service Architecture

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.

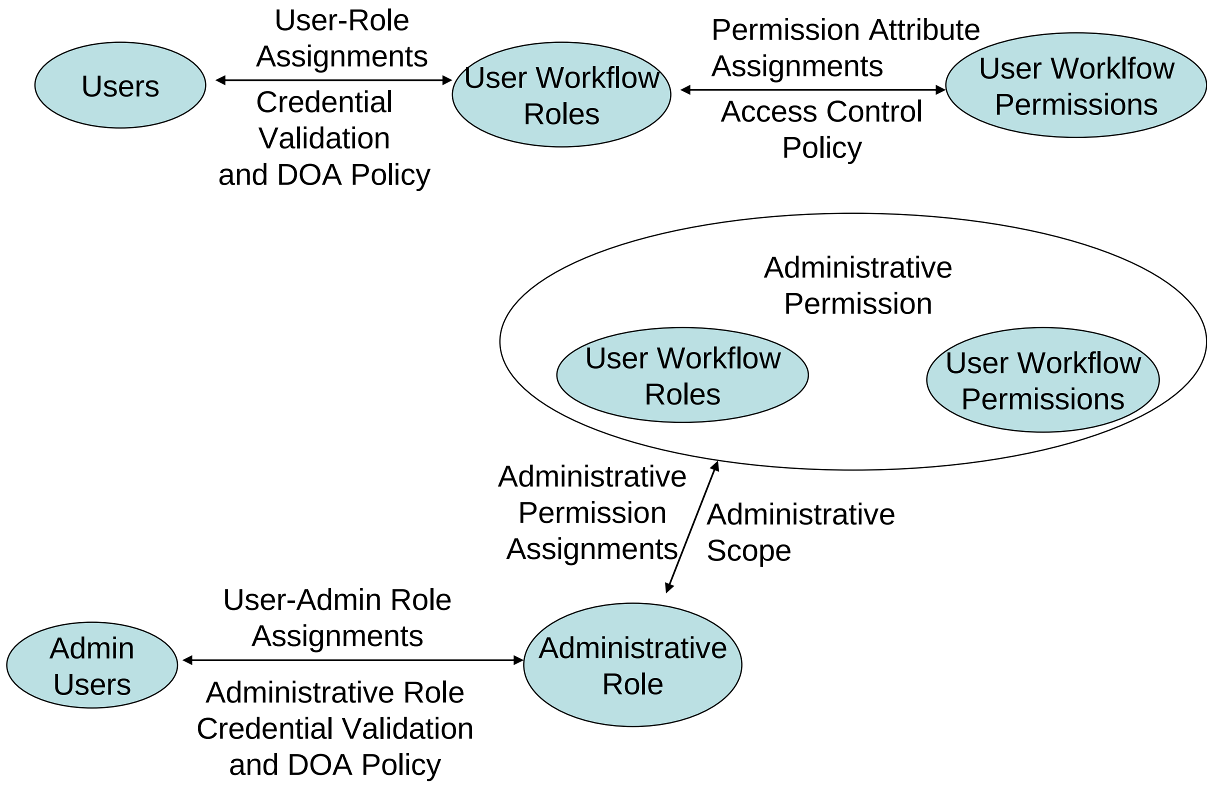

Users
User-Role Assignments
Credential Validation and DOA Policy
User Workflow Roles
Permission Attribute Assignments
Access Control Policy
User Worklfow Permissions
Administrative Permission
User Workflow Roles
User Workflow Permissions
Administrative Permission Assignments
Administrative Scope
User-Admin Role Assignments
Admin Users
Administrative Role Credential Validation and DOA Policy
Administrative Role

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).

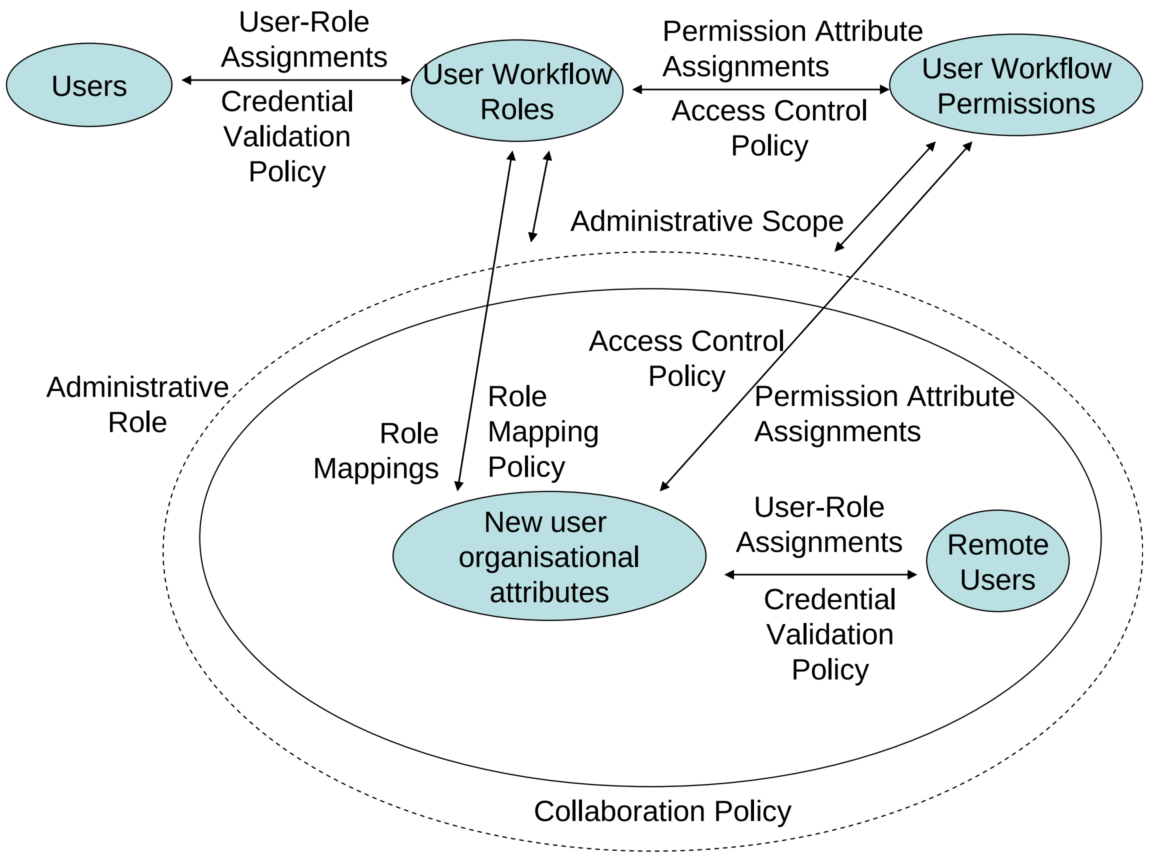
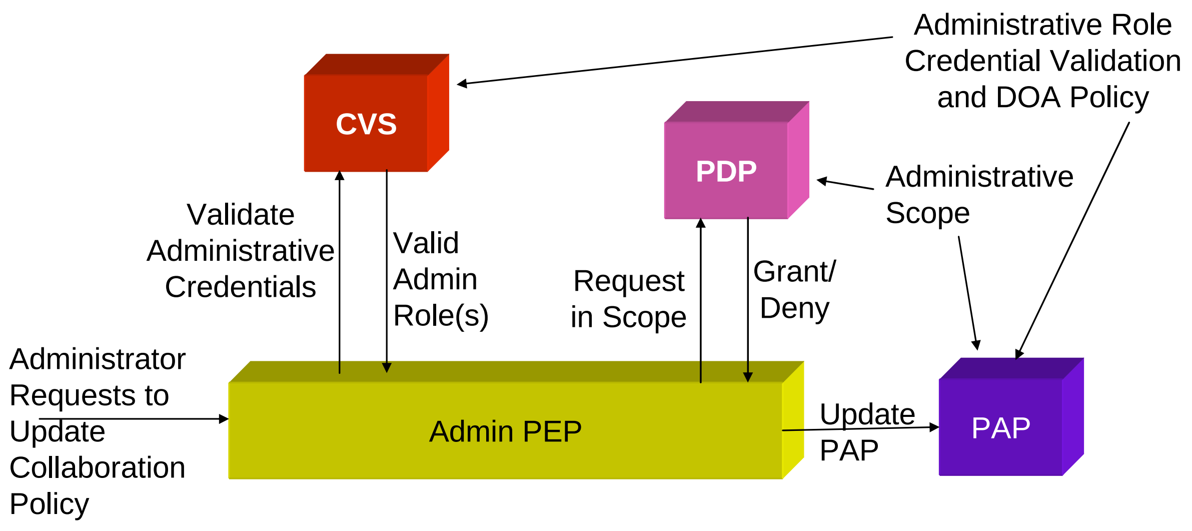


Figure 7.2. Collaboration Policies

7.1 Updating the Collaboration Authorisation Policy


Administrative Role Credential Validation and DOA Policy
CVS
PDP
Administrative Scope
Validate Administrative Credentials
Valid Admin Role(s)
Request in Scope
Grant/ Deny
Administrator Requests to Update Collaboration Policy
Admin PEP
Update PAP
PAP

CVS=Credential Validation Service
PDP= Policy Decision Point
PEP= Policy Enforcement Point
PAP= Policy Administration Point

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 have introduced the concept of sticky policies[13] [19] into the TAS^3 policy based authorisation infrastructure. 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 (see Appendix 4). Obligations are used to return or refer to the sticky policy, as obligations must be enforced by the PEP (or AIPEP) 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. This section describes how this is achieved.

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 PEP and a PEP to the PDP. The functionality of the AIPEP is to process and enforce the sticky

[13] 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.

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 8.6), whilst in the back channel model it transports the policies (see Figure 8.7). 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.

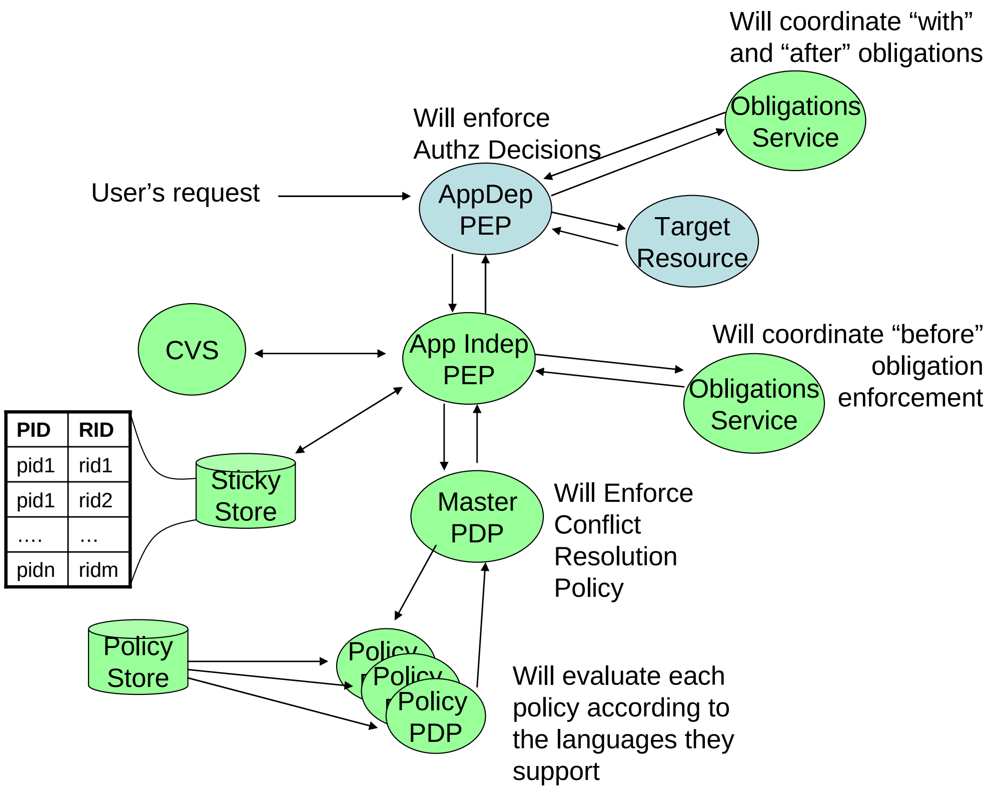

Will coordinate "with" and "after" obligations
Obligations Service
Will enforce Authz Decisions
User's request
AppDep PEP
Target Resource
CVS
App Indep PEP
Will coordinate "before" obligation enforcement
Obligations Service
PID | RID
pid1 | rid1
pid1 | rid2
.... | ...
pidn | ridm
Sticky Store
Master PDP
Will Enforce Conflict Resolution Policy
Policy Store
Policy PDP
Will evaluate each policy according to the languages they support

Figure 8.1. The Local Sticky Policy Enforcement Infrastructure

The credential validation service (CVS) has already been described in section 6.4, the Obligations Service in section 2.5 and the Master PDP in section 5.1. The Policy Store, which is an enhancement of the policy administration point (PAP) in XACML, is the place where policies can be both stored and retrieved by the TAS^3 authorisation infrastructure. When created, each policy is given a globally unique ID - the Policy ID (PID) - by its creator, and this is used to refer to the policy in the policy store. We do not dictate the form of this identifier, but it could be a URL, URN, or object identifier. The authorisation infrastructure treats it as a globally unique string.

Each resource (e.g. PII) that is to be protected by a sticky policy is given a locally unique identifier by the PEP, which we call the resource ID (RID). We do not dictate the form of this identifier, it could be a URL, a database unique key etc. but it must be unique within the target domain of the organisation. It does not need to be globally unique. Different organisations may use different RIDs to refer to the same resource as it is passed between them. The Sticky Store is conceptually a very simple component which keeps a binding of PIDs to RIDs. This is a many to many binding, in that one policy can control access to many resources and one resource can be controlled by many policies.

8.1 User input of PII and sticky privacy policy

The user is presented with an application dependent GUI and is asked to enter their PII. This requires existing GUIs to be enhanced to invite the subject to choose or enter their privacy policy. We do not specify how this is done. Organisations may have a limited number of options that a user can choose from e.g. a set of purposes, retention periods, audit requirements etc., or they may provide the user with the ability to enter their own fully specified privacy policy encoded in some standard policy language such as XACML or EPAL. The TAS^3 infrastructure is not constrained in the privacy policy languages that it can support, subject to the availability of an appropriate PDP that can evaluate authorisation decision requests and return an authorisation decision response via the standard interface (the XACML request-response context).

When the application server receives the user's input, it must extract the subject's privacy policy from the application layer message and pass this to the AIPEP along with an authorisation decision request "can this user submit this PII (with this unique RID) to the data store, using this policy in conjunction with the existing policies". In this way the application does not need to understand the contents of the subject's privacy policy since the authorisation infrastructure will handle it in an application independent way. The AIPEP takes the policy, stores it in the policy store and is returned the PID of the policy. It then constructs a new PDP that can process this policy, passing it either the policy or the PID depending upon how the PDP is constructed. The AIPEP has a manifest that records the number of PDPs that are currently active, along with their PIDs. These include the PDPs that were initialised when the authorisation infrastructure was started, plus any additional PDPs that have been dynamically constructed since then. The size of the manifest is an implementation configurable parameter depending upon the CPU and memory of the processing machine. Once the PDP has been constructed the AIPEP passes the authorisation decision request to the Master PDP along with the set of PDPs that should be used to process this request.

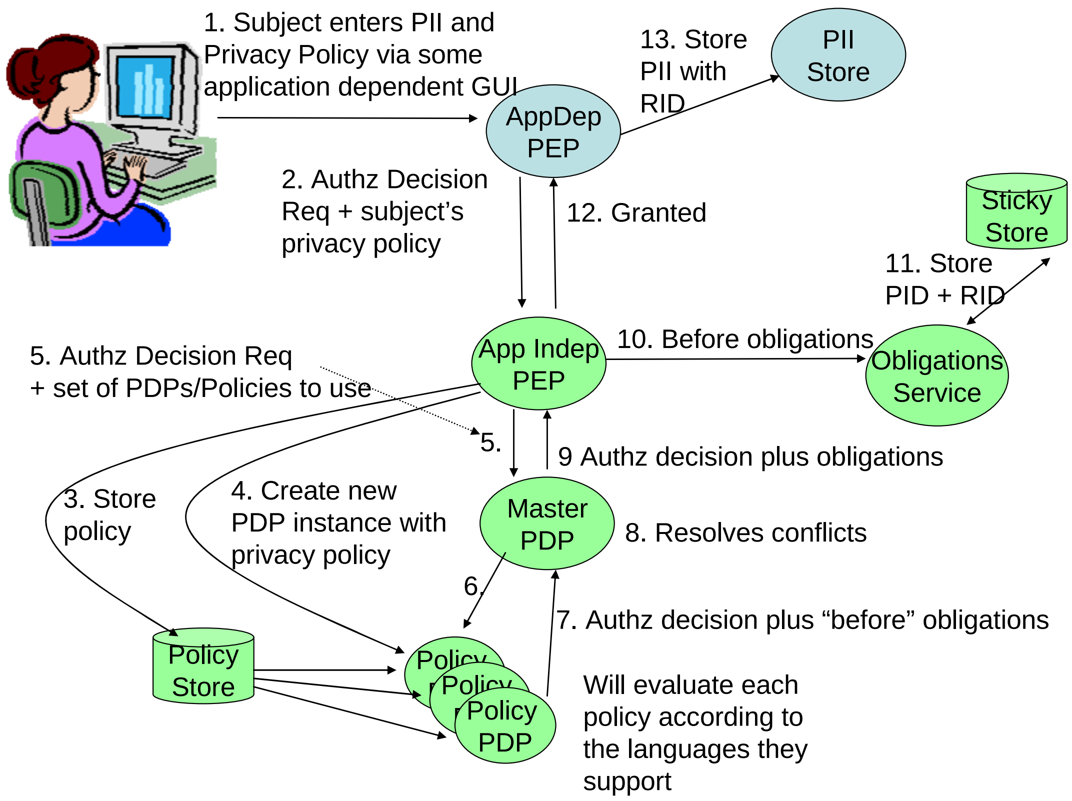

1. Subject enters PII and Privacy Policy via some application dependent GUI
AppDep PEP
13. Store PII with RID
PII Store
2. Authz Decision Req + subject's privacy policy
12. Granted
Sticky Store
11. Store PID + RID
5. Authz Decision Req + set of PDPs/Policies to use
App Indep PEP
10. Before obligations
Obligations Service
5.
9 Authz decision plus obligations
3. Store policy
4. Create new PDP instance with privacy policy
Master PDP
8. Resolves conflicts
6.
7. Authz decision plus "before" obligations
Policy Store
Policy
Policy
Policy PDP
Will evaluate each policy according to the languages they support

Figure 8.2. User input of PII plus privacy policy

The Master PDP consults its conflict resolution policy to determine how to resolve the authorisation decisions from the multiple PDPs. The Master PDP then calls the subordinate PDPs, either sequentially (first applicable rule) or in parallel (override rules) and analyses the returned decisions according to the rule. If the CRR is deny or grant overrides, and there is more than one such response of the same type, then the obligations from all such similar responses are combined together in the response that is returned to the AIPEP by the Master PDP. If a grant is returned this will contain at least one "before" obligation which instructs the authorisation system to store the subject's sticky policy in the sticky store.

The AIPEP calls the obligations service passing it the set of received obligations. This obligations service is only configured to process "before" type obligations, one of which will be instructions to the sticky store obligation service to store the subject's sticky policy. Other "before" type obligations may be configured into any of the policies of the subordinate PDPs, such as "audit the authz decision" etc. Only if all "before" type obligations are successfully enacted will the AIPEP return granted to the PEP. If any of the obligations fail to be enacted, then the AIPEP will return a deny response and will rollback its actions i.e. remove the subject's privacy policy from the policy store, terminate the appropriate subordinate PDP and remove it from its manifest. When the PEP receives the granted response it will store the user's PII in its application dependent storage.

8.2 Transfer of PII to a Third Party

When a third party wishes to transfer the subject's PII to its site, the application PEP intercepts the client's request and makes an authorisation decision query to the authorisation infrastructure asking if this remote system has permission to transfer (or other access mode depending upon the client's request) the PII identified with this unique RID. The AIPEP queries the sticky store to ask which sticky policies are bound to the resource with this RID. The sticky store returns the set of policy PIDs that are applicable.

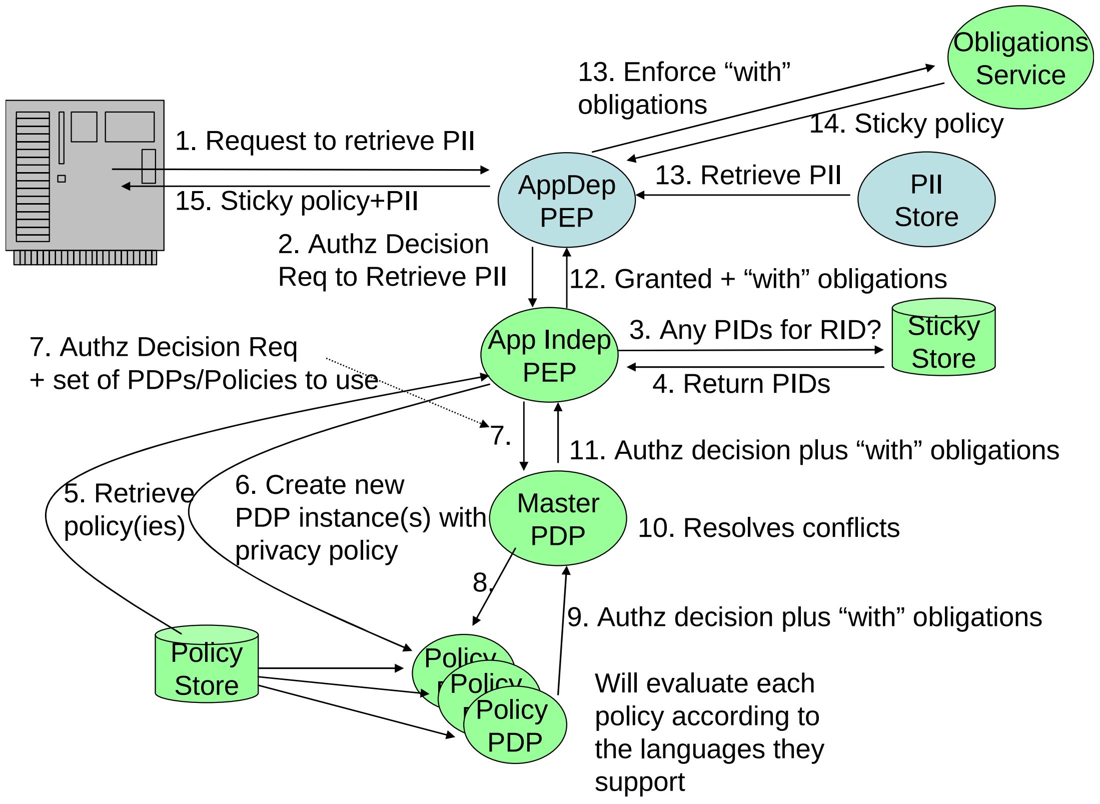

Obligations Service
13. Enforce "with" obligations
14. Sticky policy
1. Request to retrieve PII
15. Sticky policy+PII
AppDep PEP
13. Retrieve PII
PII Store
2. Authz Decision Req to Retrieve PII
12. Granted + "with" obligations
7. Authz Decision Req + set of PDPs/Policies to use
App Indep PEP
3. Any PIDs for RID?
4. Return PIDs
Sticky Store
7.
11. Authz decision plus "with" obligations
5. Retrieve policy(ies)
6. Create new PDP instance(s) with privacy policy
Master PDP
10. Resolves conflicts
8.
9. Authz decision plus "with" obligations
Policy Store
Policy PDP
Will evaluate each policy according to the languages they support

Figure 8.3. Transfer of PII to a Third Party

The AIPEP consults its manifest to see which of these policies are currently loaded into PDPs, and if some are not, retrieves the appropriate policies from the policy store and initialises the corresponding subordinate PDPs. It then passes the authorisation decision request to the Master PDP along with the set of PDPs to query. The Master PDP consults its conflict resolution policy to determine which strategy to use (ordered or unordered) and passes the authorisation decision request to the subordinate PDPs either sequentially or in parallel. The subordinate PDPs are asked if the third party is allowed to retrieve the PII. Along with a grant decision will be a "with" obligation to transfer all the sticky policies along with the PII. This obligation will be passed up to the PEP (APE model), which is now required to enact this obligation simultaneously with retrieving the subject's PII from its data store, or will be enforced by the AIPEP (ESL and backchannel models). With the APE model, the only modification that is needed to the application code is to call the obligations service, which has a uniform interface regardless of the obligations that need to be enforced. The sticky policy obligation handling service has a callback to the PEP which allows it to put the sticky policy in the relevant protocol field of the application message, ready for transfer to the third party.

Once the third party receives the response to its request to retrieve the PII, the protocol flow is now very similar to that described in section 8.1 i.e. a user submitting new PII with a sticky policy. The application PEP extracts the sticky policy and passes this to the authorisation infrastructure along with the authorisation decision request "can this client receive this PII (with this unique RID) into its local data store, using this policy in conjunction with the existing policies". Providing the local legal and keeper's policies either grants permission with a CRR of deny overrides (or are silent on this issue), then the data subject's policy will be enforced as directed by the subject's sticky policy. The data subject's policy (at least) will require the third party to store and enforce the subject's policy, and this will cause a before obligation to be returned to the AIPEP, which will ensure that the sticky policy is safely stored in the application independent infrastructure before returning grant to the application PEP. If the authorisation infrastructure is not able to enforce the sticky policy obligation then the application PEP will be denied permission to receive the PII.

Appendix 4 provides examples of the protocol message exchanges between the PEP and AIPEP. The next section describes the alternative methods for passing the sticky policy and data between application services in different systems.

8.3 The Application Protocol Enhancement (APE) 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 8.6), and passes this to the AIPEP for an authorisation decision.

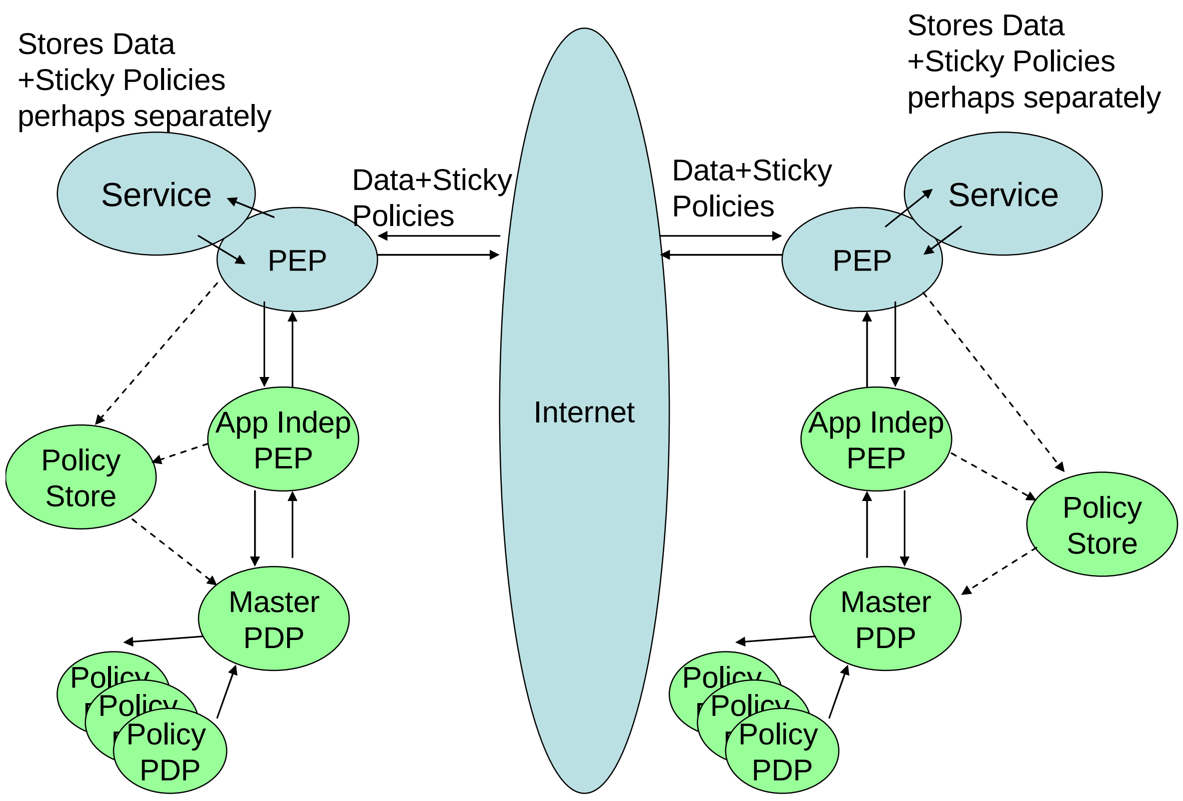


Figure 8.4. The Application Protocol Enhancement Model

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[14]. If this is Grant, the Master PDP returns an "attach sticky policy" obligation to the AIPEP saying that sticky policies should accompany the outgoing message segment. The AIPEP enforces this obligation via its local ObligationsService, and the obligation handling service extracts the policy(ies) from the policy store and gives them to the AIPEP via a callback mechanism. The AIPEP inserts the policies into a "sticky policies" obligation in the SAML-XACML authorisation decision response along with the permis authorisation decision, and sends

[14] 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.

this to the PEP (see Appendix A4.4). The PEP must then enforce this obligation, which via another callback will allow the PEP to copy the sticky policy 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 or it could use the StickyPAD construct described in Figure 5.2. 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 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. We will use one of the above APE approaches in TAS^3.

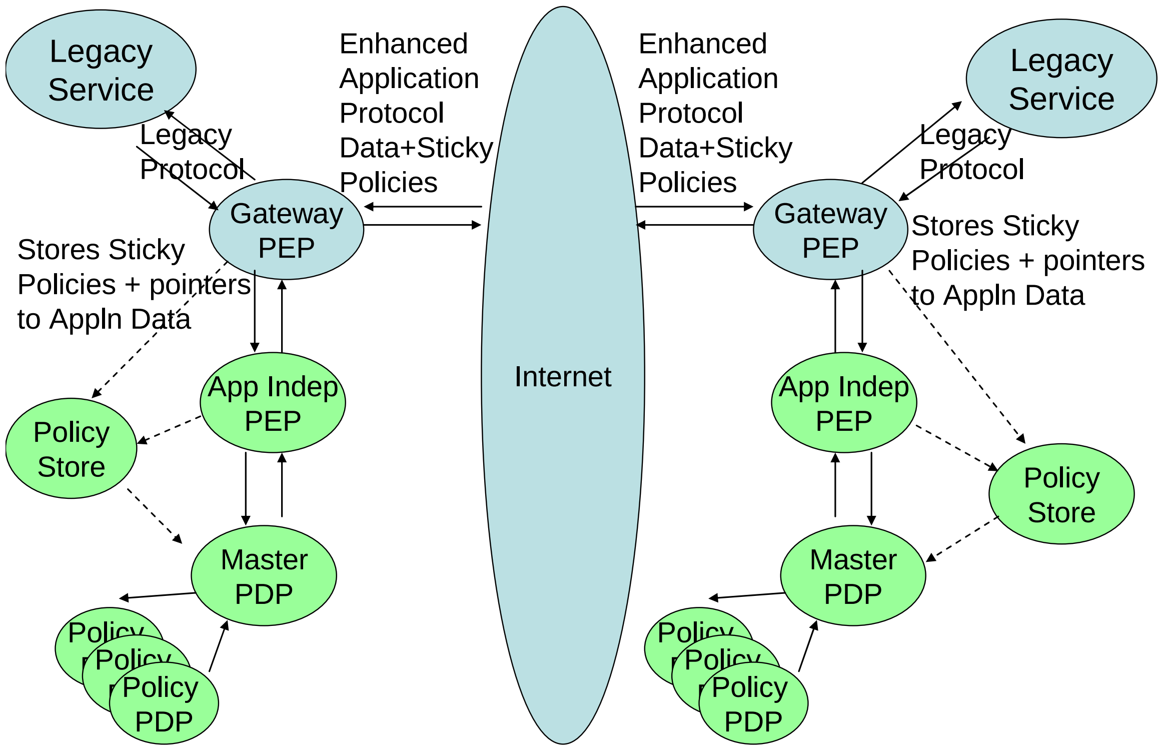

Legacy Service
Legacy Protocol
Enhanced Application Protocol Data+Sticky Policies
Internet
Enhanced Application Protocol Data+Sticky Policies
Legacy Service
Legacy Protocol
Gateway PEP
Stores Sticky Policies + pointers to Appln Data
Gateway PEP
Stores Sticky Policies + pointers to Appln Data
App Indep PEP
Policy Store
App Indep PEP
Policy Store
Master PDP
Master PDP
Policy PDP
Policy PDP

Figure 8.5. 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 8.5). 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 sticky policy(ies), message segments and relevant information for the PDP (see Section 8.6) and passes these to the AIPEP. The AIPEP parses the sticky policy and processes it. 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. 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. 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).

8.4 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/TLS approach, only in this case the security layer is responsible for carrying policies between systems rather than MACs and encrypted messages. Note that the ESL approach is not being adopted in TAS^3. It is described here for completeness only, as it was one of the methods initially considered.

In more detail the system is proposed to work as follows. The PEP parses the outgoing message that is to be enforced, segments and extracts relevant information from it (see Section 8.6), 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 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 sticky 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.

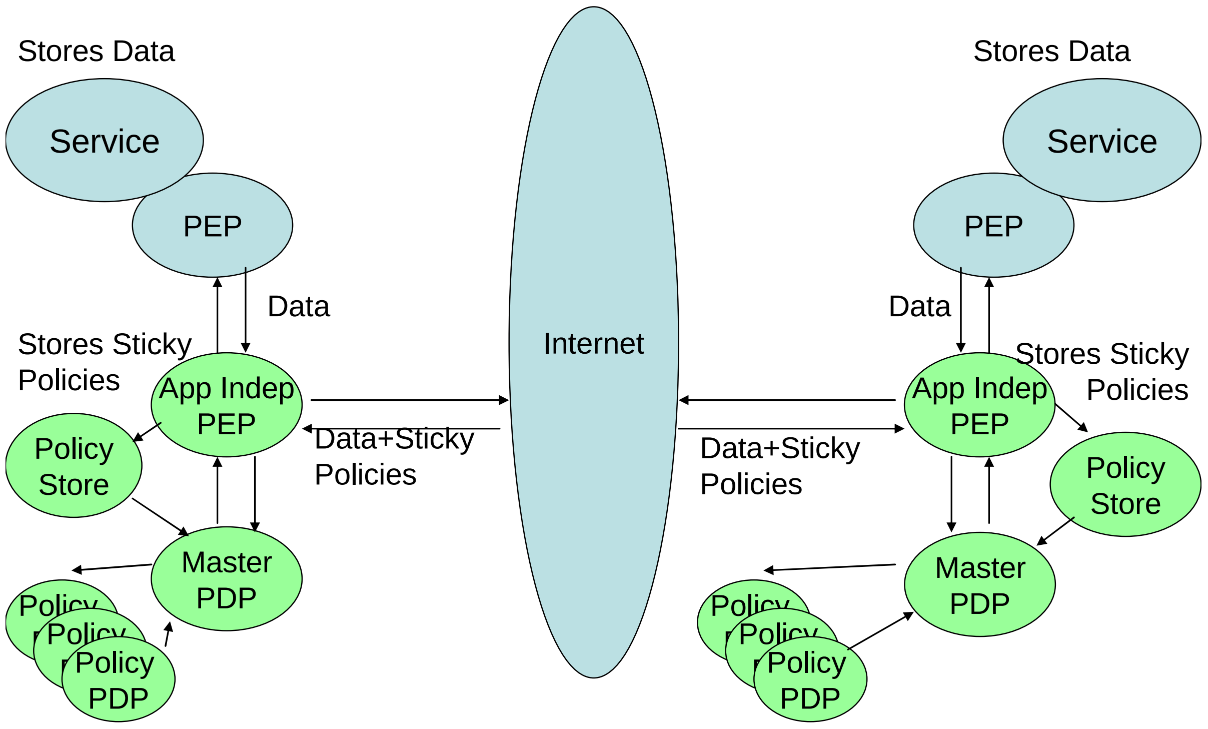


Figure 8.6. The Encapsulating Security Layer Model

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.

8.5 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.

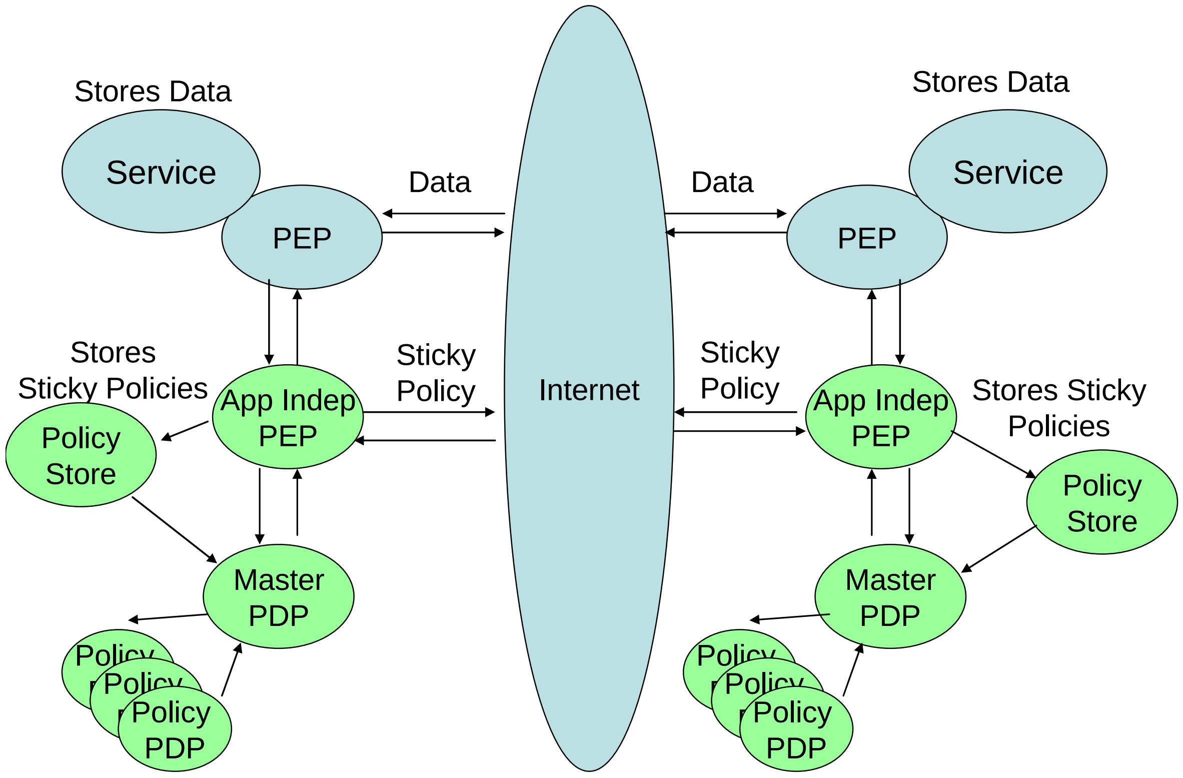


Figure 8.7. The Back Channel Model

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 8.7) 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 8.6).

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 stores incoming sticky policies in its policy store so that they can be linked to their data segments/request contexts, through the sticky store, 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.

8.6 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 TAS^3 project we 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.

8.7 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 incorporates 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 is used by the TAS^3 infrastructure to send messages about specific security, privacy and trust related events. It is called the Audit Bus. 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. We propose to use this mechanism to update sticky policies that have been distributed throughout a trust network.

Another 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 is given in D8.2 [32].

10 Authorization Ontology

10.1 Overview

This section describes the design of an ontology model for authorization policies and a way of using an ontology service to aid authorisation decision making. The ontology model is related to the common ontology suggested in WP2 (in deliverables D2.2 [48] and D2.3 [49]). The common ontology is defined in a declarative and generic way, so it cannot be used to represent concrete access control policies. The common ontology defines the lexicon used across the whole TAS3 project. In this document, we define an ontology for representing concrete policies by using this lexicon in the context of access control policies.

Figure 10.1 shows the top layer of the common ontology. The main assumption is that the world (or Thing) 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 an Entity, a Predicate or another Descriptor.

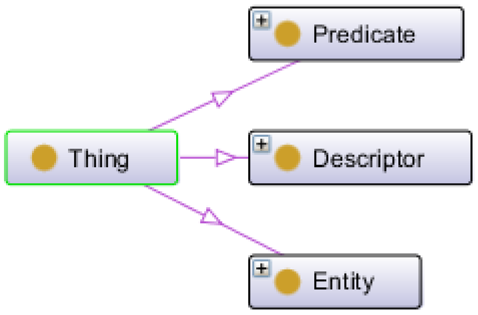


Figure 10.1. The top layer of the Descriptive Upper Ontology

Figure 10.2 show the major sub classes of Entity. The Entity class has two direct sub classes: Agent and Object. Agents represents entities that perform actions, whilst Objects represent entities that can have actions performed on them. Please note: an individual can be an individual of more than one class. Persons are agents when they are subjects in access control policies who perform actions on protected resources. Persons can also be objects when they are the subject of data protected in a privacy policy, which will be introduced later in this section. AuthoritySource is a special type of Agent who can assign security attributes used in access control policies.

Resource, UserDomain and SecurityPolicy are objects, which can be managed and accessed by other Agents in the system.

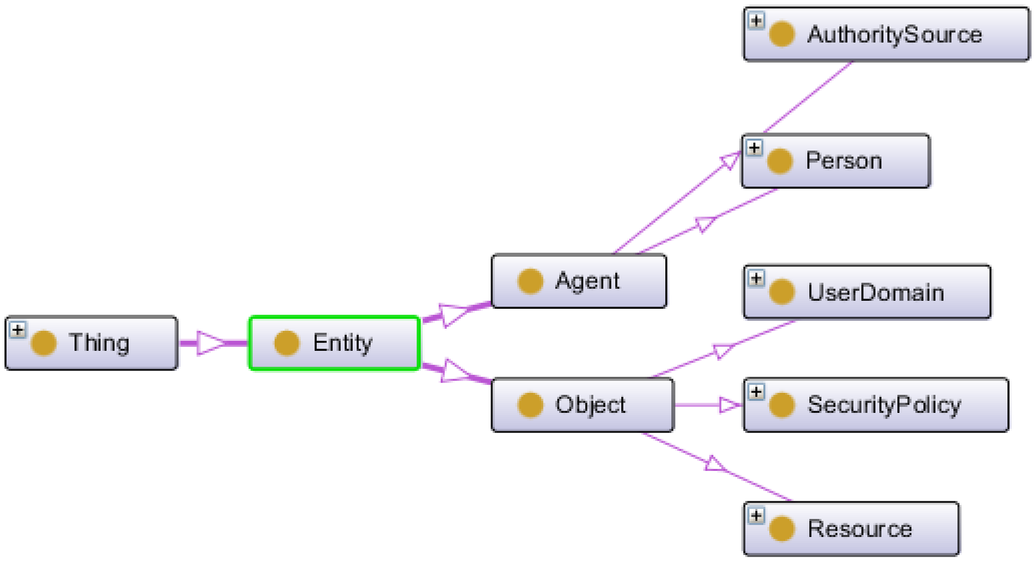


Figure 10.2. Entity

According to D2.3, the Event sub type of the Predicate type is used in the ontology to represent an access control policy. Action is the class of all actions that can be performed in the system governed by the access control policy. Decision is the outcome from a PDP regarding an access control request. This can be used to trigger Obligations defined in an ObligationPolicy.

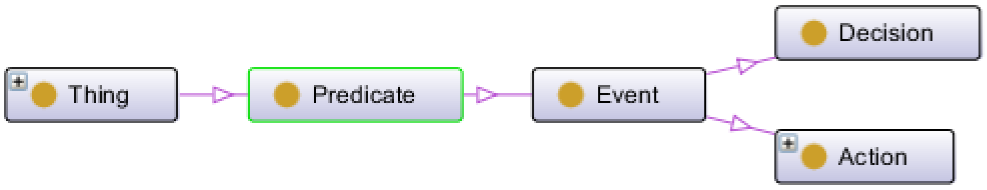


Figure 10.3. Predicate

Descriptors are classes which describe Entities and Events in a system that is governed by an access control policy. These classes are shown in Figure 10.4. Time is used to represent the validity period of an attribute or the temporal type of an Obligation; Parameter is used to describe actions defined in a policy; Purpose states the purpose of an action; Obligation is used to describe obligations triggered in an ObligationPolicy; Permission represents permissions associate with a role or other attributes. Identifier is used to identify entities in the policy, it could be an URI, an email address or an LDAPDN; Location is used to describe location information associated with an individual; Attributes here are specific for attributes used in ABAC policies, SecurityRole is a sub type of it used in RBAC policies. Conditions specifies constrains of a permission or a policy.

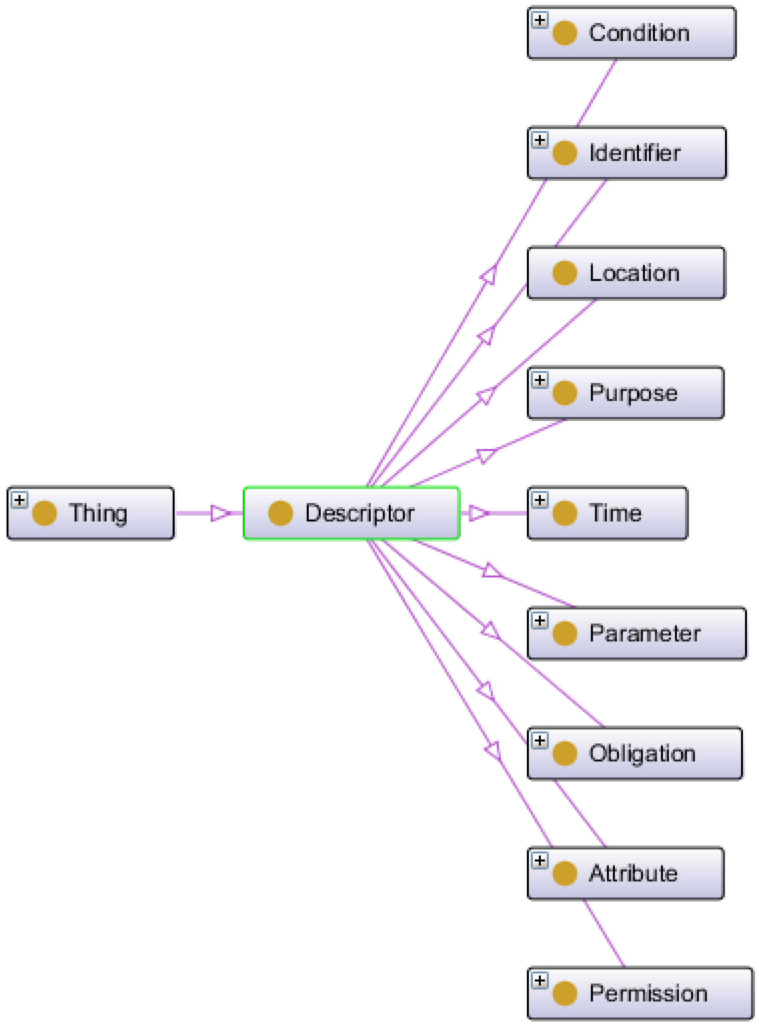


Figure 10.4. Descriptor

SecurityPolicy is the top level of policies. There are 4 sub types of it: AccessControlPolicy is used to control access to protected resources; AttributeAssignmentPolicy is used to control and validate attribute assignment in a system. PrivacyPolicy is used to state privacy protection rules, such as how long personal data can be retained; ObligationPolicy states obligations triggered by certain events, such as when access is granted. A SecurityPolicy may contain other SecurityPolicies as sub policies.

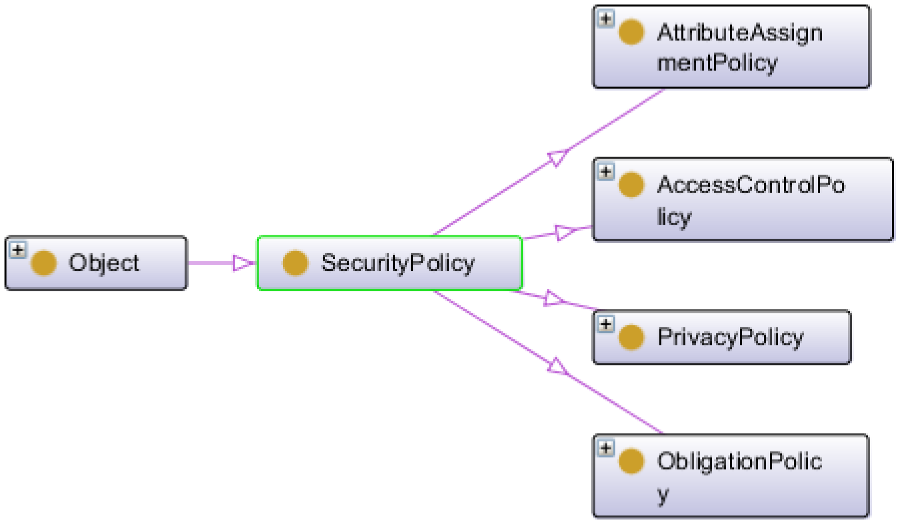


Figure 10.5. SecurityPolicy

Different types of access control policy can be defined as sub types of AccessControlPolicy. In this ontology model, we defined ABACPolicy for policies based on the ABAC model. RBACPolicy is defined as a more specific type of ABAC policy, which constrains the attribute to SecurityRole. A role hierarchy can be defined by an RBACPolicy. A set of Permissions are associated with each SecurityRole. A User with a SecurityRole attribute will be granted the associated permissions defined in the policy. RoleAssignmentPermissions in RoleAssignmentPolicy say how these roles are issued and validated.

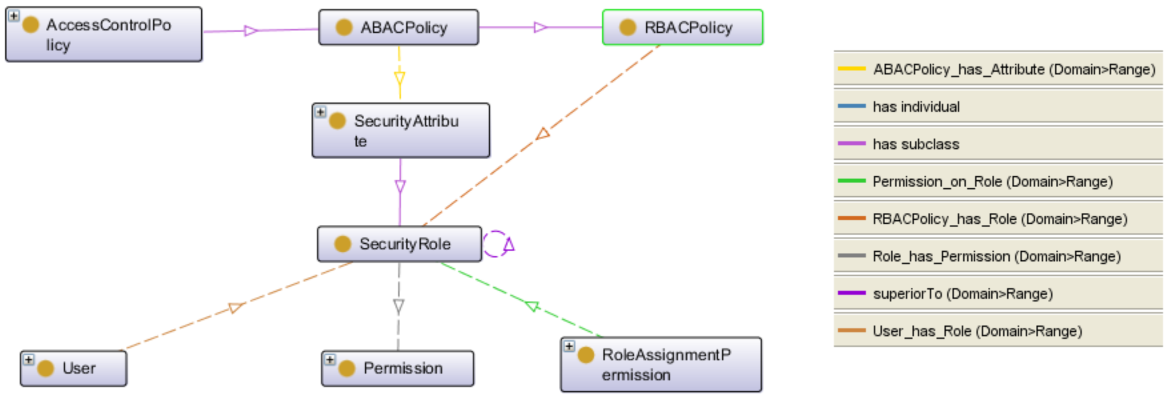

AccessControlPolicy
ABACPolicy
RBACPolicy
SecurityAttribute
SecurityRole
User
Permission
RoleAssignmentPermission
ABACPolicy_has_Attribute (Domain>Range)
has individual
has subclass
Permission_on_Role (Domain>Range)
RBACPolicy_has_Role (Domain>Range)
Role_has_Permission (Domain>Range)
superiorTo (Domain>Range)
User_has_Role (Domain>Range)

Figure 10.6. RBACPolicy

RoleAssignmentPolicy is a type of SecurityPolicy which controls how SecurityRoles are issued and validated. A RoleAssignmentPolicy defines a set of Administrators and the RoleAssignmentPermissions associated with these administrators. Each RoleAssignmentPermission is a sub type of Permission. It states a set of SecurityRoles and to which (users from which) UserDomains these roles are allowed to be assigned to.

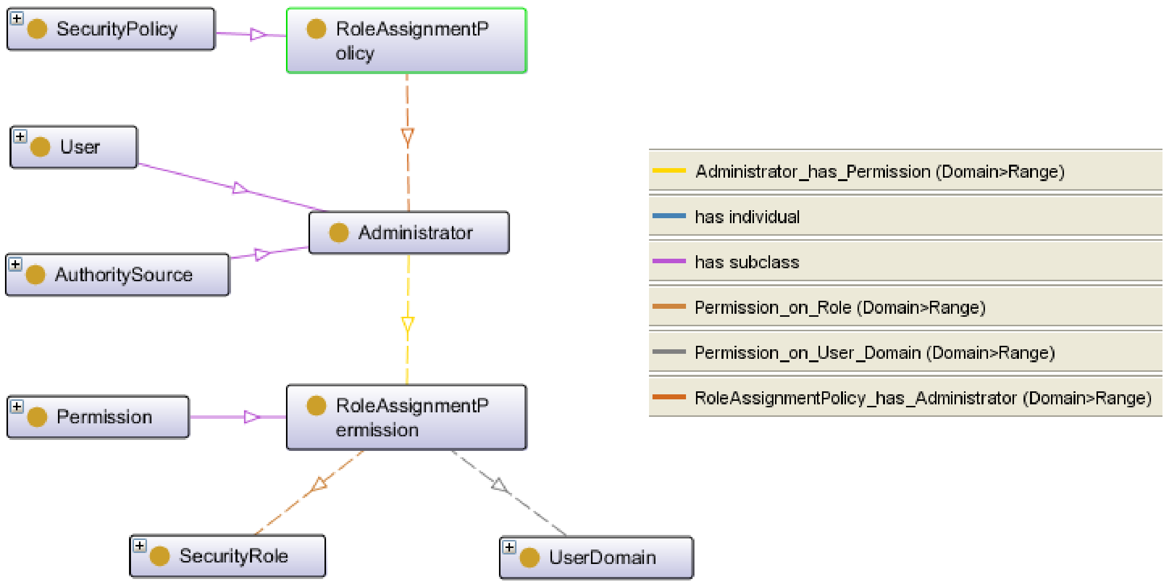

SecurityPolicy
RoleAssignmentPolicy
User
Administrator
AuthoritySource
Permission
RoleAssignmentPermission
SecurityRole
UserDomain
Administrator_has_Permission (Domain>Range)
has individual
has subclass
Permission_on_Role (Domain>Range)
Permission_on_User_Domain (Domain>Range)
RoleAssignmentPolicy_has_Administrator (Domain>Range)

Figure 10.7. RoleAssignmentPolicy

PrivacyPolicy is a type of SecurityPolicy protecting the PersonalData of users. It specifies the constraints of Purpose and retention period that have to be applied to Actions performed on the PersonalData. Also, a PrivacyPolicy may require consent of one or more specified Person, which in most cases is the subject of the Data.

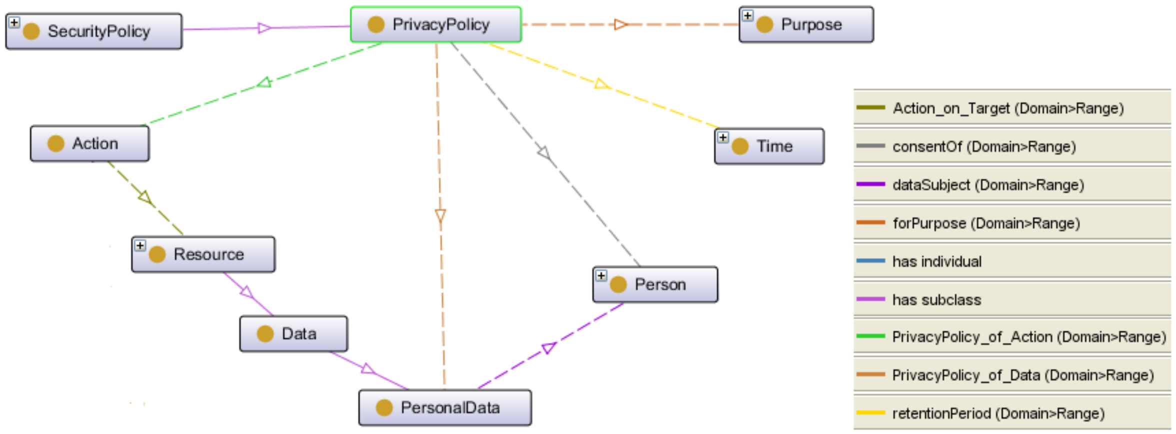

SecurityPolicy
PrivacyPolicy
Purpose
Action
Time
Resource
Person
Data
PersonalData
Action_on_Target (Domain>Range)
consentOf (Domain>Range)
dataSubject (Domain>Range)
forPurpose (Domain>Range)
has individual
has subclass
PrivacyPolicy_of_Action (Domain>Range)
PrivacyPolicy_of_Data (Domain>Range)
retentionPeriod (Domain>Range)

Figure 10.8. PrivacyPolicy

ObligationPolicy defines a set of Obligations, which are triggered by a grant or deny decision about an Event. The TemporalType specifies at which stage of the Event the Obligation should be performed viz: Before, With or After the Event. An Obligation can be associated with another Obligation as its fallback; if the later fails during its enforcement the former replaces it. EMailObligation is an example obligation which sends an email when a certain event occurs. In this example it is associated with two Email properties, which are the From and To fields of the Email. Properties for the subject and the body of the email could also be defined for an EMailObligation.

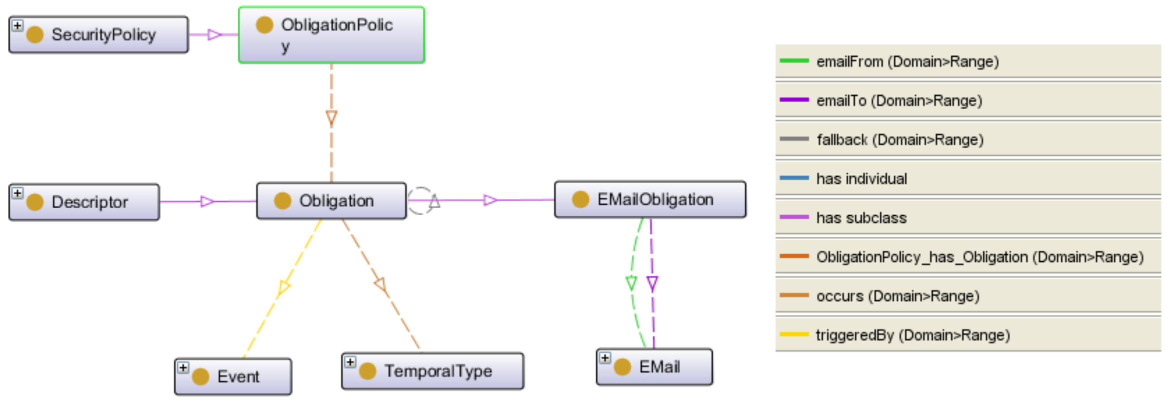


Figure 10.9. ObligationPolicy

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 10.10. The description of each class is given below.

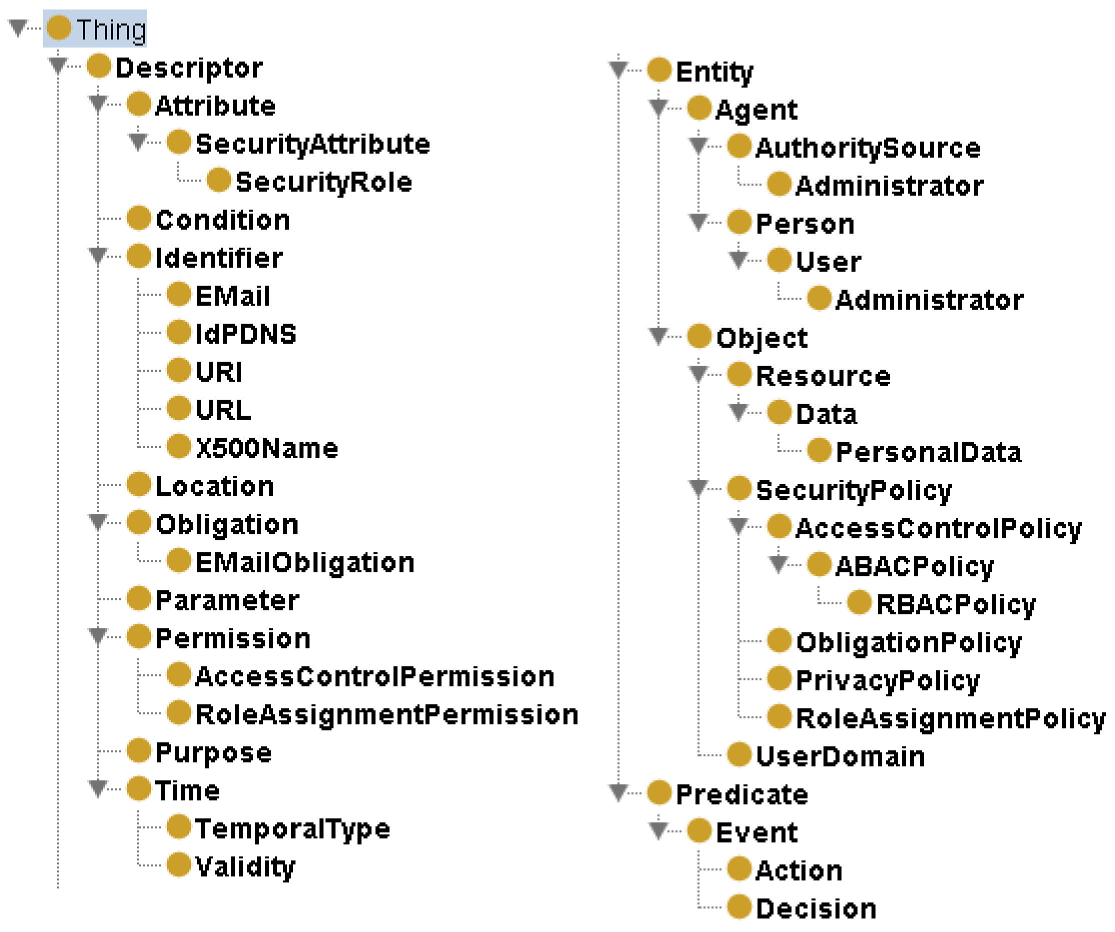


Figure 10.10. The Authorisation Ontology Class Hierarchy

ABACPolicy
An access control policy based on the ABAC model.
Properties:
ABACPolicy_has_Attribute: The SecurityAttributes covered by this policy.

AccessControlPermission
A Permission allowing an agent to perform an Action on a Resource.
Properties:
Permision_of_Action: The action which is allowed to be performed.
Permission_on_Resource: Specifies on which resource the Action is allowed to be performed.

AccessControlPolicy
An access control policy determines who is authorized to access a resource.

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.

Administrator
Administrators are a kind of User, who are an AuthoritySource at the same time. They are trusted to issue SecurityRoles to be used in RBAC policies.
Properties:
Administrator_has_Permission: This defines the RoleAssignmentPermission associated with an Administrator.

Agent
Imported from common ontologies defined by WP2.

Attribute
Imported from common ontologies defined by WP2.

AuthoritySource
The source of authority which is trusted to issue SecurityAttributes.

Condition
Condition is a boolean expression, which can be evaluated as TRUE or FALSE. Conditions can be used to constrain permissions in a policy.

Data
Data is a type of Resource, which can be further used by the requester.

Descriptor
Imported from common ontologies defined by WP2.

Decision
The outcome of evaluation an access request by a PDP.

EMail
An Email address.

EmailObligation
EmailObligation specifies an obligation that an Email should be sent out in conjunction with the Event corresponding to 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.

Event
Imported from common ontologies defined by WP2.

Identifier
Identifier is used to identify an Entity in the policy.

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

Location
Imported from common ontologies defined by WP2.

Object
Imported from common ontologies defined by WP2.

Obligation
Obligation is an operation that should be performed in conjunction with the enforcement of an event, which is the subject of an authorization Decision.
Properties:
fallback: specifies another obligation that should be enforced if the current one fails.
fulfillOn: This specifies the Obligation should be performed in conjunction with which type of Decision.
occurs: specifies at which stage of the enforcement of an Event, the Obligation should be performed. There are three possible values of temporalType: Before, With or After.

ObligationPolicy
An ObligationPolicy defines a set of Obligations. It can be used alone to specify a set of Obligations that are triggered by an event in the system; or it can be used together with an AccessControlPolicy to be triggered by an access decision.
Properties:
ObligationPolicy_has_Obligation: the Obligations defined by the policy.

Parameter
Parameters of an Action.

Permission

A Permission represents an Agent which is allowed to perform an Action on a certain Entity.
Properties:
Permission_on_Condition: The Condition constrains the Permission.

Person
Imported from common ontologies defined by WP2.

PersonalData
PersonalData is a type of Data regarding a Person. PersonalData can be protected by a PrivacyPolicy.
Properties:
dataSubject: The subject Person of the data.

Predicate
Imported from common ontologies defined by WP2.

PrivacyPolicy
The Privacy constraints on a personal Data resource.
Properties:
PrivacyPolicy_on_Data: To which Data resource this privacy policy applies.
PrivacyPolicy_on_Action: What actions this policy applies to.
consentOf: Specifies the person who must provide consent before the data is used.
forPurpose: For what purpose a Data resource can be used.
rententionPeriod: For how long the data can be retained by the requester.

Purpose
Purpose represents the reason for an action on a resource.

RBACPolicy
An access control policy based on the RBAC model.
Properties:
RBACPolicy_has_Role: The hierarchy of SecurityRole defined in this policy. Each role is associated with a set of permissions. Users with a SecurityRole attribute will be granted the permissions associated with the role.

Resource
Resources are those entities on which Actions can be applied.

RoleAssignmentPermission
A Permission for assigning a SecurityRoles to [people from] a UserDomain.
Properties:
Permision_of_Role: The SecurityRole that is allowed to be assigned.

Permission_on_UserDomain: Specifies to which [people from which] UserDomain the SecurityRole is allowed to be assigned.

RoleAssignmentPolicy
RoleAssignmentPolicy defines a set of Administrators and RoleAssignmentPermissions associated with them.
Properties:
RoleAssignmentPolicy_has_Administrator: The Administrators defined in the policy.
RoleAssignmentPolicy_has_RoleAssignmentPermission: The set of permissions for assigning SecurityRoles to [people from] UserDomains.

SecurityAttribute
Imported from common ontologies defined by WP2.

SecurityPolicy
SecurityPolicy is the top level of all types of policy. The semantic of a SecurityPolicy depends on the application consuming the policy. For example, an RBAC PDP may interpret an attribute in a different way than a standard ABAC PDP does. A SecurityPolicy may contain other policies as sub policies.
Properties:
subPolicies: The sub policies of a SecurityPolicy.

SecurityRole
SecurityRole is a type of SecurityAttribute. It is equivalent to Role in the RBAC model. A Role is associated with a set of Permissions. 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.

TemporalType
Specifies at which stage of the enforcement of an Event, the Obligation should be performed. There are three possible values: Before, With or After.

Time
Imported from common ontologies defined by WP2.

URI
URI (Uniform Resource Identifier).

URL

URL (Uniform Resource Location).

User
Users are Persons who can perform Actions in a system.
Properties:
User_has_Role: User can have the Role attribute, which is associated with a set of Permissions.

UserDomain
A group of users identified by a single Identifier.

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.

X500Name
A name in an X500 directory service.

10.3 Ontology-based Interoperability Service (OBIS)

The ontology-based interoperation service (OBIS) has the role of calculating the dominance relationship between security attributes based on the generic ontology-based data matching framework (ODMF) [50]. This service provides an automated and optimized method for the association of procedures, policies, controls and contractual obligations with data elements and roles, which makes it different from other approaches.

The ontology-based interoperation service provides semantic interoperability between service requesters and service providers based on the security policy ontology (SecPODE). The main assumptions are that (1) inputs are fully qualified URIs and (2) the service requesters and the service providers both commit to the SecPODE ontology. OBIS then determines the similarity between *subjects*, *actions*, *targets* and *environments*. More specifically, OBIS returns a value representing the relation between values within the same category.

10.3.1 Overall Architecture

OBIS is presently located in the Authorization Architecture, as shown in Figure 10.11. Similarly, OBIS could be located within any of the TAS3 services (e.g. trust negotiation service).

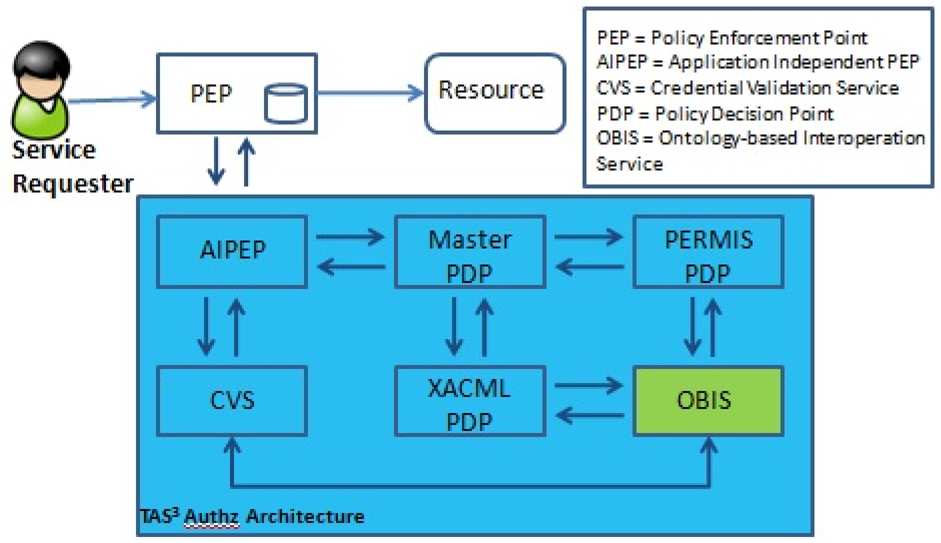

PEP = Policy Enforcement Point
AIPEP = Application Independent PEP
CVS = Credential Validation Service
PDP = Policy Decision Point
OBIS = Ontology-based Interoperation Service
Service Requester
PEP
Resource
AIPEP
Master PDP
PERMIS PDP
CVS
XACML PDP
OBIS
TAS³ Authz Architecture

Figure 10.11. OBIS integration within the TAS³ Authorisation Architecture

Based on the above architecture, the authorization process contains the following steps:

- Step 1: The Service Requester (SR) launches a request to access distributed data repositories protected by security policies; the request needs to be validated against each service provider.
- Step 2: The Application Independent Policy Enforcement Point (AIPEP) contacts the Credential Validation Service (CVS) to validate the requestor's set of credentials (issued by multiple credential issuing services).
- Step 3. If the CVS cannot validate any credentials, it contacts OBIS to determine the relationship between the presented credentials and the terms it has in its credential validation policy rules. The CVS returns the set of validated attributes to the AIPEP.
- Step 4: The AIPEP contacts the Master PDP, whose role is to call specific PDPs depending on which language the policy is expressed in (see XACML and PERMIS PDPs in Figure 10.11); the advantage of this separation is that each service provider does not have to implement the same policy language.
- Step 5: If a PDP is not able to make an access control decision, then OBIS is called to calculate the semantic relation between terms in the policy and terms in the request context, within the same category (i.e. *Subject, Role, Environment* or *Resource*).
- Step 6: Based on the values returned by the OBIS, the access request is either granted or denied.

10.3.2 OBIS Architecture

The architecture of the Ontology-based data matching system is illustrated in Figure 10.12. It contains two main modules: the translator and the path finder.

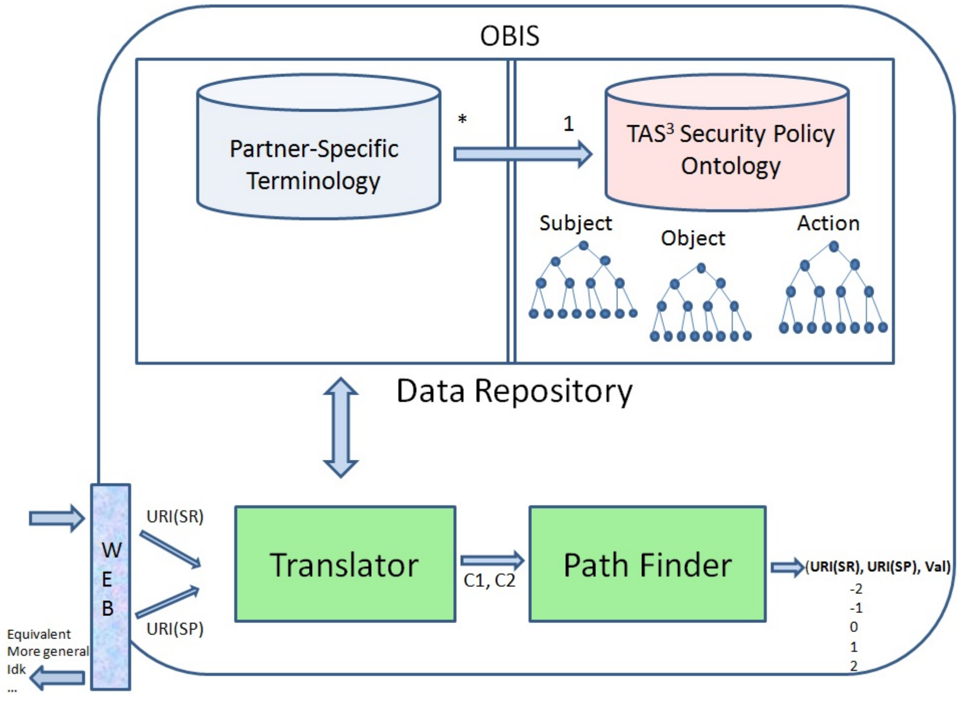


Figure 10.12. OBIS Architecture

The translator takes an input string corresponding to a term in the user's/system's terminology and maps it into a concept from the ontology. As a first step, we implemented a many-to-one mapping (i.e. many terms in the user's terminology map to one concept in the ontology). The translator uses a terminology base (data repository), where each partner has to contribute his own vocabulary (terminology). Each vocabulary is mapped to the security policy ontology. Each vocabulary-ontology mapping is regarded as a dictionary. An annotator is used to annotate the vocabulary terms with concepts from the ontology. This step is performed twice, once for a term from the security policy (T1) and once for a term from the request context (T2).

The ontology, the system and the communication channel are protected to the same degree of security as the TAS^3 Authorization architecture.

The path finder takes the output of the translator (i.e. two concepts, C1 and C2 from the ontology, corresponding to the two input strings T1 and T2) and returns the path (level of dominance) between them. There are six possible return values from the path finder: -3, -2, -1, 0, 1 and 2.

-3 means "I don'tknow T2",
-2 means "I don't know T1"
-1 means "T1 is not related to T2",
0 means "T1 is dominated by (less general than) T2",
1 means "T1 is equivalent to T2",
2 means "T1 dominates (is more general than) T2".

The path is searched in a Directed Acyclic Graph (DAG) corresponding to the Subject-Action-Object category. The path finder is exemplified in Table 10.1.

Table 10.1 – Security Term Dominance Examples.

T1	T2	Level of dominance
Read	**Delegate**	**-3**
Placement Provider	**Contact Person**	**2**
Education	**CV**	**0**
Salary	**Resource**	**0**
…	**…**	**…**

The level of dominance is calculated based on ontology-based data matching strategies, introduced in the next section.

10.3.3 Ontology-based Security Concept Matching

The security concepts level of domination is calculated using the generic ontology-based data matching framework (ODMF). Given an ontology, ODMF is able to perform the following operations:

- maps data into semantic networks (Tree, Directed Acyclic Graph / lattice or any directed graph);
- perform semantic computation, such as path recognition (shortest path, connectivity), path strength in scores (semantic vicinity, etc.), composite semantic similarity of semantic networks;
- perform literal computation, such as fuzzy similarity of literals (strings) based on known algorithms (SecondString [51], UnsmoothedJS [52][53][54], JaroWinklerTFIDF [52][53][54] and TFIDF [55]); and
- perform lexical computation, such as synonymous similarity (WordNet [56], etc.), similarity based on user dictionary.

There are several ontology-based data matching strategies in the ODMF. Each strategy contains at least one graph algorithm. In the case of OBIS, we focus on the Controlled Fully Automated Ontology Assisted Matching (C-FOAM) matching strategy. We adapted C-FOAM to compare two security terms annotated with security policy concepts.

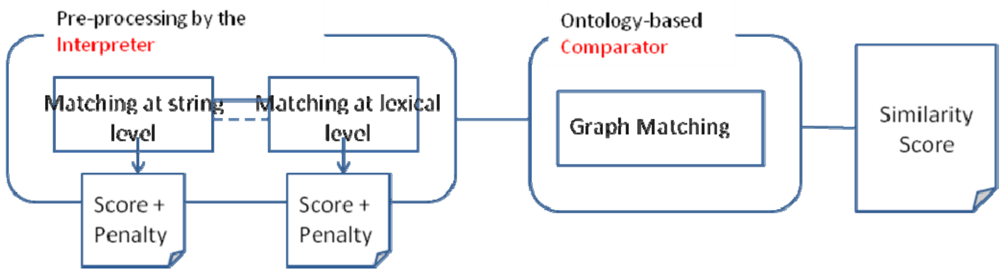


Figure 10.13. C-FOAM Model

C-FOAM contains two modules (see Figure 10.13) – the interpreter and the comparator – corresponding to the two components of OBIS. The interpreter module makes use of the lexical dictionary and the domain ontology to interpret the input. Given a term that denotes a concept in the domain ontology, the interpreter will return the correct concept defined in the ontology. The comparator then uses any combination of the different graph algorithms for the security concepts path recognition.

Due to the specificity of the security concepts (i.e. the domination relation), here we only use relations which grow or shrink a set (e.g. ‘is-a” or “part-of”). C-FOAM uses AND/OR graphs for computing the dominance relation between the two concepts identified by the interpreter.

10.3.4 OBIS Applied to an eLearning Scenario

Within the TAS^3 project, we have developed an employability demonstrator focusing on the management of internships and work placements for university students [57]. Timely, accurate and secure presentation and exchange of verified skills data and personal information is key to the success of this scenario. For example, recruiters and prospective employers want to access verified data in a standardized format (e.g. HR-XML) to facilitate comparisons and matching of students with job profiles. Similarly, candidates want to regain control over how their personal information is accessed, processed and stored by third party by setting security policies.

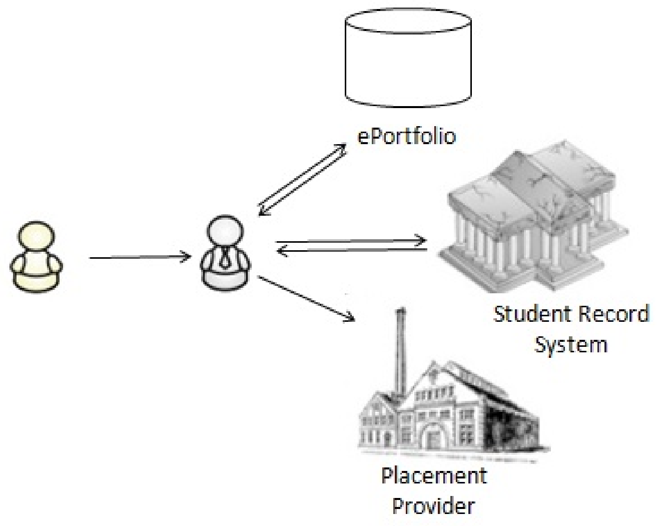


Figure 10.14. The Employability Scenario

Figure 10.14 represents one of the employability scenarios in TAS^3. In this scenario, Alice is a second year student at a UK university and seeks a summer work placement. Alice contacts a placement service approved by the university to discuss the details of her application. Her placement advisor, called Bob, informs her that he first needs to verify that she is a registered student at the university. Once Bob has received the confirmation, he contacts Alice to get permission to access relevant information to match her to available placements. Alice is happy to share this information subject to this information not being shared to third parties without her approval. Based on this information, Bob identifies a number of placement providers that he believes have suitable placements for students like Alice. Alice wishes to be put forward for two placements and agrees that the placement advisor can act on her behalf to agree terms of a work placement and she consents to have relevant personal information to be disclosed to them. Bob forwards Alice's information to the placement providers by email for consideration.

In this scenario, several security and trust issues may be encountered, as follows:

- Does the student trust that the placement advisor is approved by the university?
- Can the student trust that only relevant personal information is used during the placement process?
- Can the student trust that the information provided to the placement provider is protected as per her privacy policies?
- The placement advisor, placement providers, etc. use their own systems to store the information. How can all stakeholders be sure that personal information is secure between one placement and another?

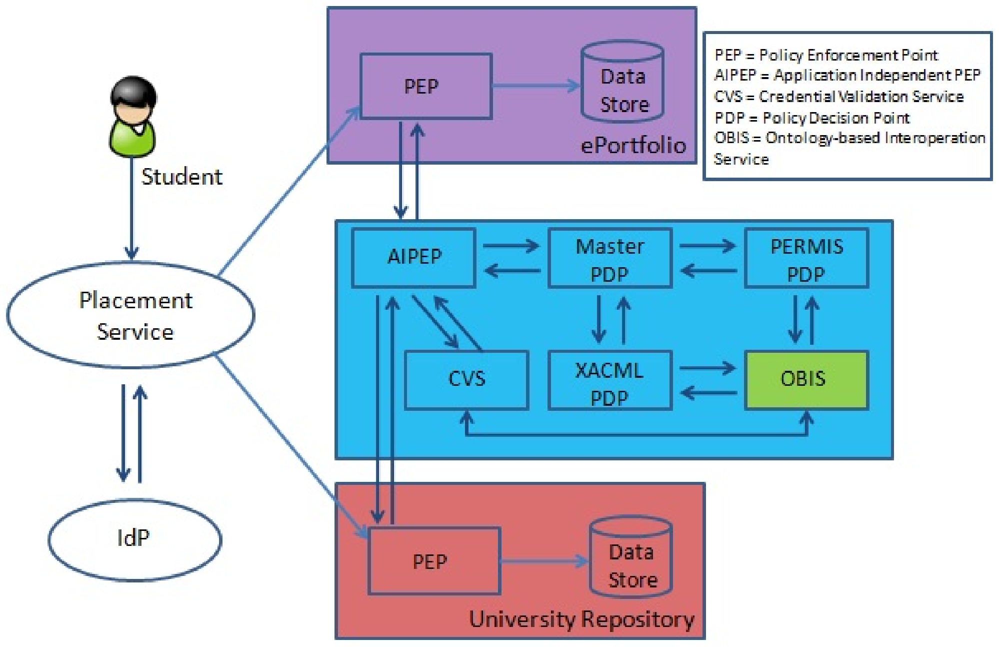

PEP
Data Store
ePortfolio
PEP = Policy Enforcement Point
AIPEP = Application Independent PEP
CVS = Credential Validation Service
PDP = Policy Decision Point
OBIS = Ontology-based Interoperation Service
Student
Placement Service
IdP
AIPEP
Master PDP
PERMIS PDP
CVS
XACML PDP
OBIS
PEP
Data Store
University Repository

Figure 10.15. Architecture for the enforcement of security and privacy policies

Figure 10.15 shows a high level conceptual model for the enforcement of security and privacy policies in the employability scenario. When a person issues a request for a placement or internship, the placement service first verifies with the Identity Provider (IdP) that the person is who she claims to be. For instance, the placement service would contact the university to check whether the person (e.g. Alice) is a registered student.

After the authentication process, Bob needs to access relevant information about Alice from distributed data repositories (e.g. University ePortfolio and CV repository). The information stored in these repositories is protected by security policies set up by Alice. For instance, Alice writes **a privacy policy stating that only members of the university have permission to access her work in the university ePortfolio**. Thus, Bob's request needs to be validated against each service provider to see whether he can gain access to Alice's personal information.

OBIS's role in the employability scenario is illustrated with some examples in Table 10.2.

The security policy ontology used for this use case scenario is partially depicted in Figure 10.16, with the focus being on the *Resource*.

Table 10.2 – OBIS's Role in Authorisation Interoperability.

OBIS Role	Security Concepts Matching
To check whether Bob can access Alice's personal data by calculating the relation between the placement advisor (Bob) **role** in the request context and the University Staff role in Alice's policy (to determine whether a foreign role dominates the local role in the credential validation policy).	**Role** matching for CVS. **Translator**: T1 = Placement Advisor T2 = University Staff **Path finder: 1** "equivalence" ➔credential is valid
To determine whether the **resource** to be accessed by the service requester (e.g. Alice's previous employment experience) is more specific than the resource protected by the security policy (i.e. Alice's e-Portfolio).	**Resource** matching for PDP. **Translator**: TI = Alice previous employment experience T2 = Alice's e-Portfolio **Path finder: 0** "less general" ➔Access granted

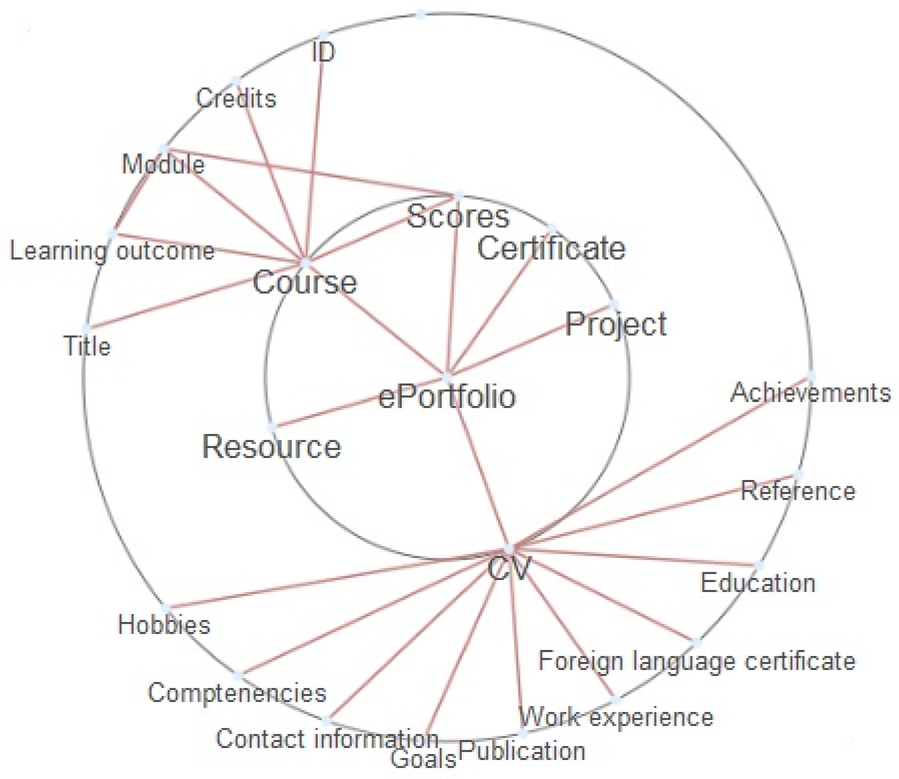


Figure 10.16. A Part of the Security Policy Ontology (focus on Resource).

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 the relevant policies are not 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[15];

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

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-centric 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

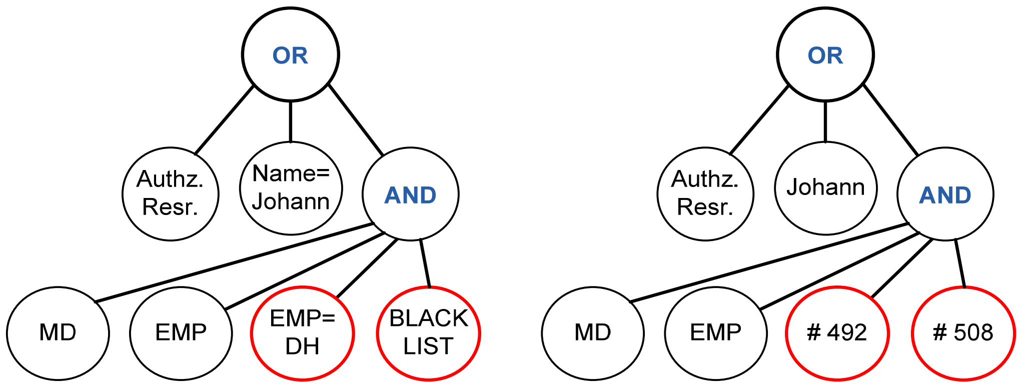
This section describes, on a conceptual level, the policies used for TAS3 trust negotiation as well as the negotiation. Details on policy language, negotiation strategies and further implementation details are given in the section 11.3.

We decided to use the UniPro approach [63] 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 [63] 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 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[16]. 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.



[16] 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).

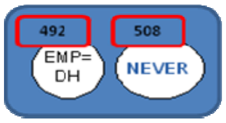


Figure 11.2 - Policies protecting resources #492 and #508

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 [63]; they are instead enhanced versions 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 a “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.

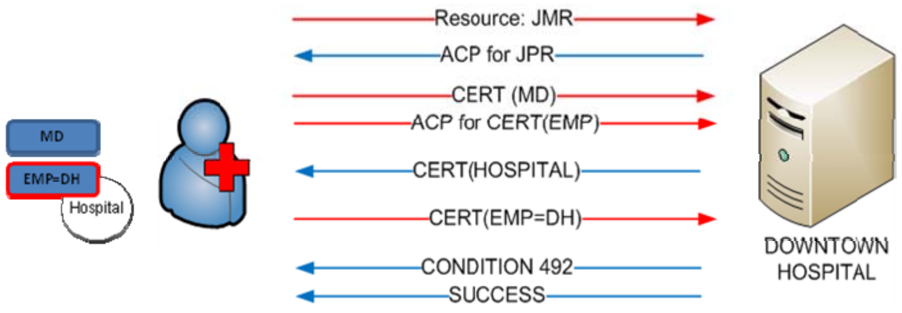


Figure 11.3 - Example negotiation message exchange

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*[17], 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.

[17] 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.

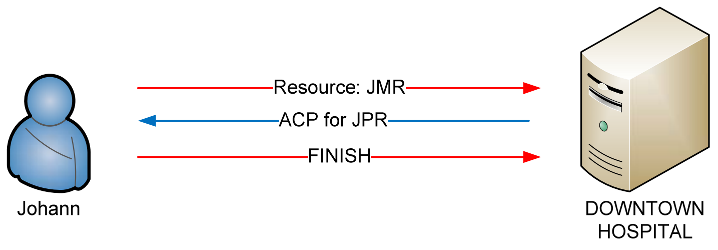


Figure 11.4 - Example negotiation message exchange

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 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 [60] with an enhanced version of the UniPro approach [63].

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 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.

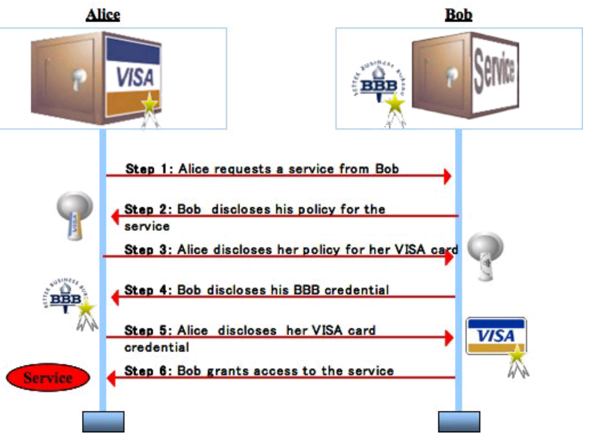

Alice
Bob
VISA
BBB
Service
Step 1: Alice requests a service from Bob
Step 2: Bob discloses his policy for the service
Step 3: Alice discloses her policy for her VISA card
Step 4: Bob discloses his BBB credential
Step 5: Alice discloses her VISA card credential
Step 6: Bob grants access to the service
Service
VISA

Figure 11.5 - TrustBuilder2 exchange[18]

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 [62] and the software itself [61].

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

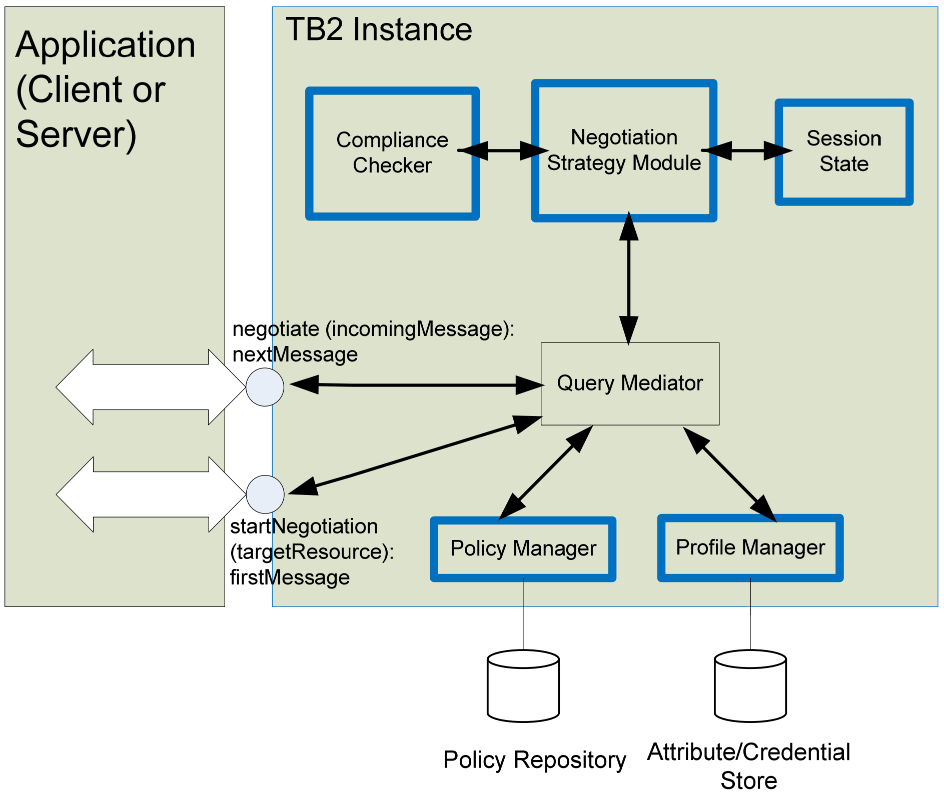

Application (Client or Server)
TB2 Instance
Compliance Checker
Negotiation Strategy Module
Session State
negotiate (incomingMessage): nextMessage
Query Mediator
startNegotiation (targetResource): firstMessage
Policy Manager
Profile Manager
Policy Repository
Attribute/Credential Store

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)[19].

[19] 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, compatible negotiation

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

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 privacy protected 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 by 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

This chapter describes two integrations of CUP trust negotiation. The first integration is into the Demonstrator Prototype of KU.Leuven, and the second one is into the TAS3 Architecture as an independent web service. For the first version of this deliverable, trust negotiation was not integrated at all. For the second version of this deliverable, the trust negotiation, evaluation and enforcement subsystem as described in the previous chapter has been integrated in the Prototype, as described in the next subsection. The third version of this deliverable also contains the integration of trust and privacy negotiation into the TAS3 Architecture, as an independent web service.

12.1 Integration into the KUL Demonstrator Prototype

12.1.1 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.

12.1.2 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.

12.1.3 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.

12.1.4 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.

12.1.5 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.

12.1.6 Policy Editor

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

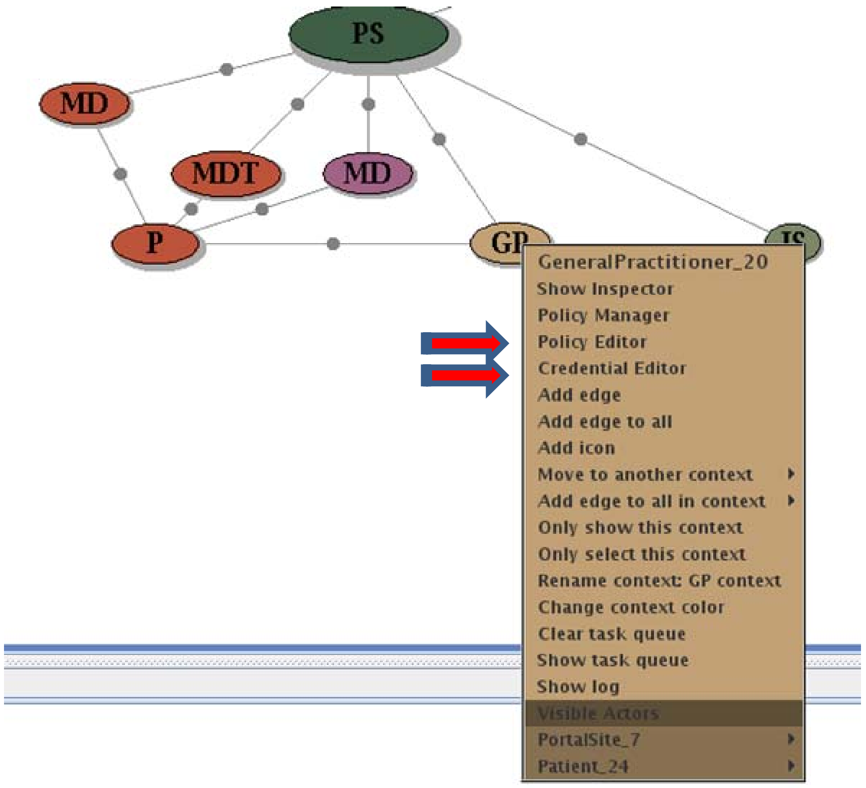


Figure 12.1 Context-sensitive menu (right-click)

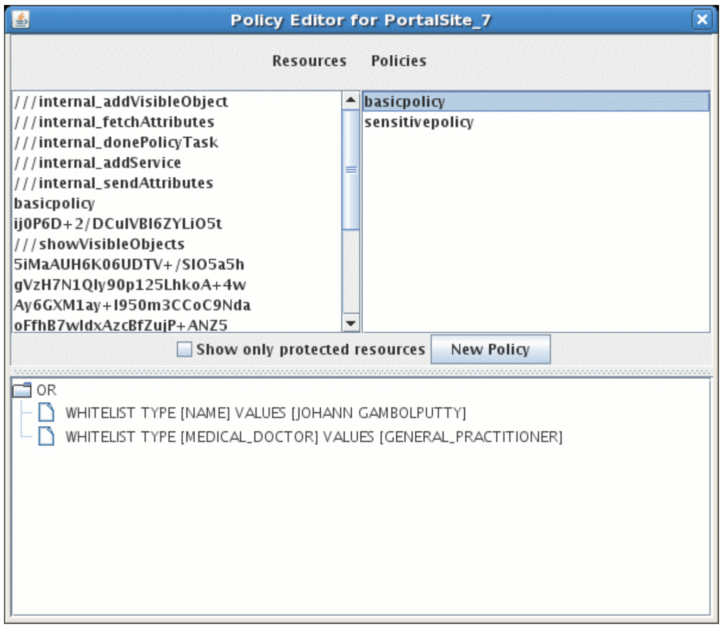

Policy Editor for PortalSite_7
Resources
Policies
///internal_addVisibleObject
///internal_fetchAttributes
///internal_donePolicyTask
///internal_addService
///internal_sendAttributes
basicpolicy
ij0P6D+2/DCulVBI6ZYLiO5t
///showVisibleObjects
5iMaAUH6K06UDTV+/SIO5a5h
gVzH7N1Qly90p125LhkoA+4w
Ay6GXM1ay+I950m3CCoC9Nda
oFfhB7wldxAzcBfZujP+ANZ5
basicpolicy
sensitivepolicy
Show only protected resources
New Policy
OR
WHITELIST TYPE [NAME] VALUES [JOHANN GAMBOLPUTTY]
WHITELIST TYPE [MEDICAL_DOCTOR] VALUES [GENERAL_PRACTITIONER]

Figure 12.2 - Policy Editor

Figure 12.2 depicts the main policy editor GUI, as shown when invoked via the right-click menu of Figure 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.

12.1.6.1 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 12.3.

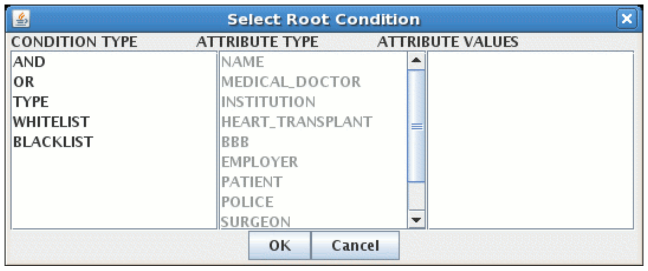


Figure 12.3 - Creating a condition

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.

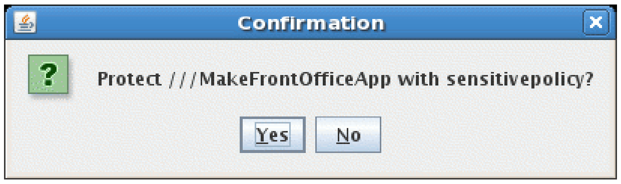
12.1.6.2 Resources

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 a 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.

12.1.6.3 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 12.4) and, if confirmed, will store the association. If the resource was already protected by another policy, this association will be overwritten.



Protect ///MakeFrontOfficeApp with sensitivepolicy?

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.

12.1.7 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 12.1, the GUI shown in Figure 12.5 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.

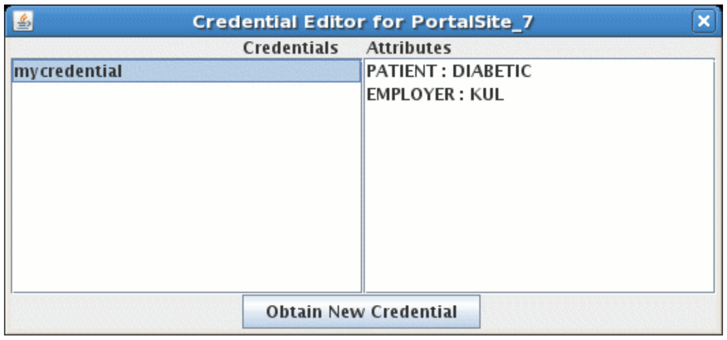


Figure 12.5 - Obtaining credentials

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 12.6 and Figure 12.7. There, the user should select the attribute types that he wishes to be included in the credential.

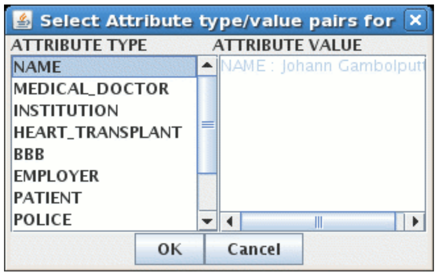


Figure 12.6 - New credential composition

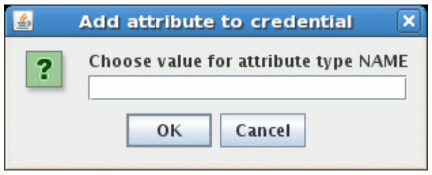

Add attribute to credential
Choose value for attribute type NAME
OK
Cancel

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 12.7 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.

12.2 Trust and Privacy Negotiation as an independent web service

This section describes the integration of the TPN client as an independent web service.
Transforming the TPN client into a web service enables a more diverse set of deployment scenarios. The web service may be running locally on the user's computer. This corresponds to

the usual deployment where users manage their credentials locally. However, as a web service, TPN can now be offered as a third party service. In TAS3, it is part of the infrastructure and therefore is offered as a TAS3 service.

TPN, as part of the TAS3 infrastructure, executes negotiation on behalf of the user. For this, the user's disclosure policies must be stored at the TPN provider, and also his credentials must be available at the time of executing the protocol. There are multiple modes of operation of the TPN web service within the TAS3 Architecture.

The first mode of operation that is chosen for implementation is the operation where TPN is invoked as part of service discovery. In this mode of operation, the discovery service invokes TPN whenever the user wishes to make use of a particular type of service.

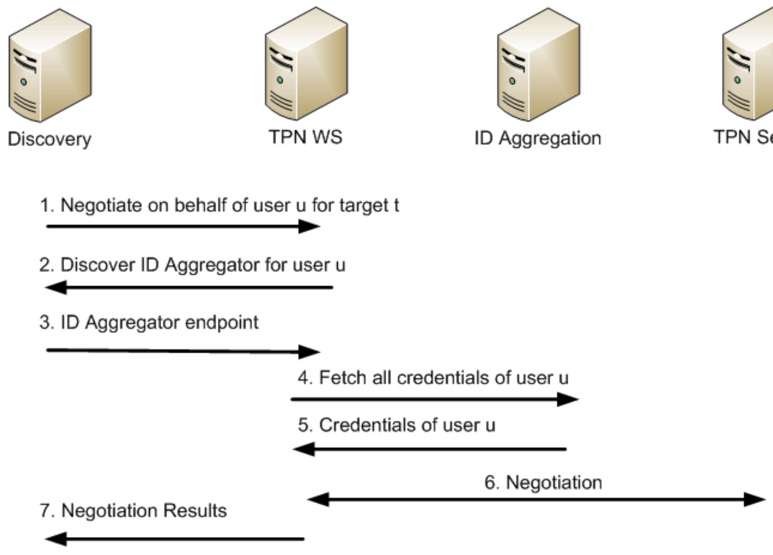

Discovery
TPN WS
ID Aggregation
TPN Server
1. Negotiate on behalf of user u for target t
2. Discover ID Aggregator for user u
3. ID Aggregator endpoint
4. Fetch all credentials of user u
5. Credentials of user u
6. Negotiation
7. Negotiation Results

Figure 12.8 - Invocation sequence with TPN web service

Figure 2.8 shows the sequence of invocations that occur when TPN is invoked in the context of the mode of operation that is chosen for implementation. The discovery service invokes TPN, typically as the result of some user action (not shown in the figure). Then, the TPN WS, which will act as the TPN client on behalf of the user, will discover the user's Identity Agregator service. This is necessary because the Aggregator service knows how to fetch the user's credentials that may have to be disclosed during negotiation. The Aggregator is resolved in steps 2 and 3. The next exchange, steps 4 and 5, actually fetches the user's credentials from the ID Aggregator. In reality, this exchange may entail multiple exchanges, because the Aggregator only returns references towards credentials which need to be resolved. For brevity, however, the credential resolution steps are not shown explicitly in the figure.

After all credentials of the user have been gathered at the TPN web service, negotiation starts in step 6. The TPN server is contact with the negotiation request for the target t (which was included in the first message in step 1. The negotiation protocol itself is not a web service protocol, but the protocol from TrustBuilder2.0.

Finally, in step 7, a report that contains the results from the negotiation is returned to the caller of the TPN webservice (in this case, the discovery service).

Many aspects of implementing TPN as a web service have been completed. This includes the specification of the protocols headers that contain the data sent between from the client that invokes the TPN webservice (step 1 in the figure), and the encoding of the negotiation result (step 7). Moreover, integrating the software with the ZXID library has been accomplished, and the invocation and parsing of the initial message (step 1) has been successfully tested.

However, at the point in time of writing this deliverable, TPN web has not yet been fully implemented as a web service. This is because some obstacles remain due to the fact that the TPN web service is an integration point where the software of multiple partners comes together and must interoperate both on the API and the protocol level. This includes the software from RISARIS (who provide the discovery service and software from KUL who provide the negotiation engine and protocol, and the software from KENT who provide the identity aggregator). Although not explicitly shown in the figure, a data structure (an endpoint reference – EPR) that is generated by the discovery service must pass through the TPN webservice (step 1) and be forwarded to the ID Aggregator (step 4) where it is consumed. This is necessary in order to correctly identify the user (although denoted by u throughout the figure, in actuality the identifier by which he is known differs between all the web services). Moreover, the credentials returned by the resolution process must be parsed by the TPN engine.

The obstacles will be overcome in the upcoming developer workshops as well as within remote developing sessions. The TPN web service is the main focus of one of the integration test scenarios (see D12.2).

13 Standardisation Activities

We have undertaken the following standardisation activity during the first three 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 been published as OGF 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 project. 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. Sampo Kellomaki attended the 78th IETF OAuth working group meeting July 25-30, 2010; Maastricht, Netherlands.
8. David Chadwick has discussed the carrying of sticky policies in any language with the OASIS XACML TC and they agreed to enhance the SAML-XACML protocol in order to allow any type of policy to be carried (and not only XAMCL policies as is specified in the original Committee Draft). This has now been published as a Committee Specification by the TC [34] and has been implemented by Kent as shown in Appendix 4. This is a major achievement for the TAS3 project.
9. David Chadwick has also discussed an enhancement to XACML to support Break the Glass policies, in order to standardise a BTG response in XACML. The first draft profile has been written by Kent, and discussed intensively in the XACML TC [59]. It is highly likely a profile will be agreed before the end of the TAS3 project.

14 Conclusions

This document represents the third phase of the design of an identity management, authentication and authorisation infrastructure for the TAS^3 project. This design has been created during the first three 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 3 of the design shows a significant enhancement over previous versions. We expect that implementation experience and more detailed technical designs in the final year of the project will further refine this design document. Integration of the IdMAA infrastructure with the application demonstrators and components from the other work packages will also serve to further inform the design. As such there is still some further work to do in the final year, and the design will be improved and modified where necessary in the light of the implementation experience and piloting in the application demonstrators.

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

a) we have designed and built the initial policy conflict resolution mechanism for the Master PDP;
b) we believe it will be relatively easy to build application dependent PEPs, as a key design aim has been to keep this as simple as possible in order to minimise the work of application developers;
c) we have designed and implemented a way to parameterise various obligations (such as send an email message and reset a BTG variable)

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

1. we have not designed the TAS^3 protocol for the encapsulating security layer described in section 8, and we do not propose to do this in the current project, since we have implemented the APE mechanism and the SOAP header as a pseudo-back channel mechanism;
2. 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. However none of the currently specified obligations do require this, and so this feature is for further study;
3. we currently do not have a mechanism for the user to update a sticky policy that has been discributed far and wide with his personal data. We are considering using the event bus for this, but this design and implementation work will only start in the final year of the project;

4. creating conflict resolution policies may prove to be too challenging for security officers. Until we carry out user trials we will not know for sure;
5. we have not yet integrated the infrastructure described here with work from all of the other WPs, such as secure repositories from WP4, the event-management bus from WP8, and the workflow engine from WP3. These may have an impact on this deliverable. However we have integrated it with the trust PDP from WP5, the authentication IDP from WP2, the trust negotiation service from KUL and an application demonstrator from WP9, so we are confident that our overall approach is correct.

15 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.

16 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] TAS^3 Architecture, TAS^3 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] TAS^3 Deliverable D8.2. "Software Documentation: System: Back Office Services", Version 2.0, due month 39

[33] TAS^3 Deliverable D4.3. "Integrated Secure Repositories", Version 1.0.

[34] OASIS. "SAML 2.0 Profile of XACML, Version 2.0". OASIS committee specification 01, 10 August 2010

[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

[50] P. De Baer, Y. Tang and R. Meersman. “An ontology-based data matching framework: use case competency-based HRM”, in Proceedings of the 4th International OntoContent Workshop, OTM 2009, Springer LNCS, pp. 514-523, Portugal, 2009.

[51] W. W. Cohen and P. Ravikumar, “Secondstring: an open source java toolkit of approximate string-matching techniques”. Project web page, http://secondstring.sourceforge.net.

[52] M. A. Jaro, “Advances in Advances in record-linkage methodology as applied to matching the 1985 census of Tampa, Florida”. Journal of the American Statistical Association 84:414–420, 1989.

[53] M. A. Jaro, “Probabilistic linkage of large public health data files”, (disc: P687-689). Statistics in Medicine 14:491–498, 1995.

[54] W. E. Winkler, “The state of record linkage and current research problems”. Statistics of Income Division, Internal Revenue Service Publication R99/04, 1999. Available from http://www.census.gov/srd/www/byname.html.
[55] J. K. Sparck, “A statistical interpretation of term specificity and its application in retrieval”, Journal of Documentation 28 (1): 11–21, 1972.
[56] C. Fellbaum, *WordNet: an electronic lexical database*, Massachusetts Institute of Technology, ISBN 0-262-06197, 1999.
[57] B. Claerhout, D. Carlton, C. Kunst, L. Polman, D. Pruis, L. Schidlers and S. Winfield. “Pilots specifications and use case scenarios”, Deliverable Deliverable D9.1, Trusted Architecture for Securely Shared Services, 2008. Available at http://bit.ly/bRBK5l.
[58] Ferreira, A., Chadwick, D., Farinha, P., Correia, R., Zhao, G., Chilro, R., Antunes, L.:How to securely break into RBAC: the BTG-RBAC model.In: Annual Computer Security Applications Conference, pp23-3,Honolulu, Hawaii, (2009)
[59] D.W.Chadwick and S.F.Lievens. “BREAK THE GLASS PROFILE FOR XACML V2.0 AND V3.0”, 25 Nov 2010.
[60] Adam J. Lee, Marianne Winslett, Kenneth J. Perano. "TrustBuilder2: A Reconfigurable Framework for Trust Negotiation." In IFIP Advances in Information and Communication Technology, Trust Management III, 176-195. Springer Boston, 2009.
[61] Database and Information Systems Laboratory, Department of Computer Science, University of Illinois at Urbana-Champaign. TrustBuilder2 Download Page. http://dais.cs.illinois.edu/dais/security/tb2/ (accessed November 19, 2009).
[62] Lee, Adam J. TrustBuilder2 User Manual. Department of Computer Science, Univ. of Illinois at Urbana-Champaign, 2007.
[63] Winslett, Ting Yu and Marianne. "A Unified Scheme for Resource Protection in Automated Trust Negotiation." IEEE Symposium Security and Privacy. IEEE, 2003.

Appendix 1. The Obligation Schema and Java Interface

The TAS^3 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 TAS^3 authorisation infrastructure, is given in section A1.2 below. The Java interface for an Obligation Service that will enforce such obligations in the TAS^3 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 TAS^3 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[20].

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.

[20] 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.

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 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.

Appendix 4. Example SAML-XACML Request-Responses with Sticky Policies

This section shows the message encodings that we have used in our first proof of concept prototype implementation to pass sticky policies between the PEP and the authorisation infrastructure. We are aware that the encodings are not yet fully standards conformant because at the time of writing there is no way to carry a policy in any language in the SAML-XACML protocol. We expect this to be rectified during the final year of the project.

A4.1 Example SAML-XACML Request to AIPEP to Submit Resource with Sticky Policy

The authorisation request shown below is requesting if a student is able to SUBMIT a resource with resource ID rid-123 to the local system (storage location not specified in this example). The SAML extension element contains the StickyPolicy that accompanies this request. The StickyPolicy contains a PERMIS authorisation policy that allows any role to SUBMIT and TRANSFER the attached resource to anywhere, but if transfer is granted the policy contains an obligation to AttachStickyPolicy to the resource on the outgoing transfer.

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:urn="urn:oasis:names:tc:xacml:2.0:profile:saml2.0:v2:schema:protocol:cd-01"
xmlns:urn1="urn:oasis:names:tc:SAML:2.0:assertion"
xmlns:xd="http://www.w3.org/2000/09/xmldsig#"
xmlns:urn2="urn:oasis:names:tc:SAML:2.0:protocol"
xmlns:urn3="urn:oasis:names:tc:xacml:2.0:context:schema:os"
xmlns:urn4="urn:oasis:names:tc:xacml:2.0:policy:schema:os"
xmlns:urn5="urn:oasis:names:tc:xacml:2.0:profile:saml2.0:v2:schema:assertion:cd-01">
  <soapenv:Header/>
  <soapenv:Body>
<XACMLAuthzDecisionQuery
xmlns="urn:oasis:names:tc:xacml:2.0:profile:saml2.0:v2:schema:protocol:cd-01"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="urn:oasis:names:tc:xacml:2.0:profile:saml2.0:v2:schema:protocol:cd-01
file:/home/sfl/work/issrg/oasis-documents/xacml3/XACML-3.0-cd-1-updated-2009-May-
07/XSD/xacml-2.0-profile-saml2.0-v2-schema-protocol-cd-1.xsd"
   ID="A2009-10-13T12.57.07"
   Version="2.0"
   IssueInstant="2009-10-13T12:58:12.209Z">
<xacml-context:Request xmlns:xacml-context="urn:oasis:names:tc:xacml:2.0:context:schema:os">
   <Subject xmlns="urn:oasis:names:tc:xacml:2.0:context:schema:os">
      <Attribute AttributeId="urn:oid:1.2.826.0.1.3344810.1.1.14"
DataType="http://www.w3.org/2001/XMLSchema#string">
         <AttributeValue>
```

```
            student
          </AttributeValue>
        </Attribute>
      </Subject>
      <Resource xmlns="urn:oasis:names:tc:xacml:2.0:context:schema:os">
        <Attribute                    AttributeId="urn:oasis:names:tc:xacml:1.0:resource:resource-id"
DataType="http://www.w3.org/2001/XMLSchema#string">
          <AttributeValue>rid-123</AttributeValue>
        </Attribute>
      </Resource>
      <Action xmlns="urn:oasis:names:tc:xacml:2.0:context:schema:os">
        <Attribute                          AttributeId="urn:oasis:names:tc:xacml:1.0:action:action-id"
DataType="http://www.w3.org/2001/XMLSchema#string">
          <AttributeValue>SUBMIT</AttributeValue>
        </Attribute>
      </Action>
      <Environment xmlns="urn:oasis:names:tc:xacml:2.0:context:schema:os"/>
</xacml-context:Request>
<samlp:Extensions xmlns:samlp="urn:oasis:names:tc:SAML:2.0:protocol">
<sp:StickyPolicy        xmlns:sp="tas3:to:be:decided:namespace"        PolicyID="sticky-policy-1"
PolicyLanguage="PERMIS"
        PolicyType="Authorization" TimeOfCreation="2010-08-09T00:00:00Z">
        <sp:PolicyAuthor                              Format="urn:oasis:names:tc:SAML:1.1:nameid-
format:X509SubjectName">cn=Stijn,ou=permisv5,c=gb</sp:PolicyAuthor>
        <sp:PolicyResourceTypes>
          <sp:ResourceType>personal:preferences</sp:ResourceType>
        </sp:PolicyResourceTypes>
        <sp:PolicyContents><X.509_PMI_RBAC_Policy
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:noNamespaceSchemaLocation="file:/media/disk/home/sfl/issrg/svn/code/trunk/build/data/ope
npermis/bundle/issrg/pba/rbac/xmlpolicy/schemachecking/policy53.xsd" OID="">
          <SubjectPolicy>
            <SubjectDomainSpec ID="everywhere">
              <Include LDAPDN=""/>
            </SubjectDomainSpec>
          </SubjectPolicy>
          <RoleHierarchyPolicy>
            <RoleSpec Type="permisRole" OID="1.2.826.0.1.3344810.1.1.14">
              <SupRole Value="UNSPECIFIED"/>
            </RoleSpec>
          </RoleHierarchyPolicy>
          <SOAPolicy>
```

```
  <SOASpec ID="anyone" LDAPDN=""/>
</SOAPolicy>
<RoleAssignmentPolicy>
  <RoleAssignment>
    <SubjectDomain ID="everywhere"/>
    <RoleList>
      <Role Type="permisRole"/>
    </RoleList>
    <Delegate Depth="0"/>
    <SOA ID="anyone"/>
    <Validity/>
  </RoleAssignment>
</RoleAssignmentPolicy>
<TargetPolicy>
  <TargetDomainSpec ID="anywhere">
    <Include RegEx=".*"/>
  </TargetDomainSpec>
</TargetPolicy>
<ActionPolicy>
  <Action Name="TRANSFER" ID="TRANSFER"/>
  <Action Name="SUBMIT" ID="SUBMIT"/>
</ActionPolicy>
<TargetAccessPolicy>
   <TargetAccess>
    <RoleList/>
    <TargetList>
      <Target>
        <TargetDomain ID="anywhere"/>
        <AllowedAction ID="SUBMIT"/>
      </Target>
    </TargetList>
  </TargetAccess>
  <TargetAccess>
    <RoleList/>
    <TargetList>
      <Target>
        <TargetDomain ID="anywhere"/>
        <AllowedAction ID="TRANSFER"/>
      </Target>
    </TargetList>
    <Obligations>
```

```
                        <Obligation
ObligationId="http://sec.cs.kent.ac.uk/obligations/AttachStickyPolicy"
FulfillOn="Permit"/>
                  </Obligations>
               </TargetAccess>
           </TargetAccessPolicy>
       </X.509_PMI_RBAC_Policy>
      </sp:PolicyContents>
   </sp:StickyPolicy>
</samlp:Extensions>
</XACMLAuthzDecisionQuery>
</soapenv:Body>
</soapenv:Envelope>
```

A4.2 AIPEP SAML-XACML Response to Request in A4.1

The SAML-XACML response message below shows that the SAML request was successful and that the XACML authorisation decision was Permit.

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">
   <soapenv:Body>
      <urn:Response                                    IssueInstant="2010-11-26T20:32:15.987Z"
ID="_a97a5ba0440fb8a1bd5a0d40c911a49b"         Version="2.0"         InResponseTo="A2009-10-
13T12.57.07" xmlns:urn="urn:oasis:names:tc:SAML:2.0:protocol">
         <urn:Status>
            <urn:StatusCode Value="urn:oasis:names:tc:SAML:2.0:status:Success"/>
         </urn:Status>
         <urn1:Assertion                                    IssueInstant="2010-11-26T20:32:15.988Z"
ID="_5fe5a3725f54b6225db89cf7fde58767"                                                   Version="2.0"
xmlns:urn1="urn:oasis:names:tc:SAML:2.0:assertion">
            <urn1:Statement                    xsi:type="urn:XACMLAuthzDecisionStatementType"
xmlns:urn="urn:oasis:names:tc:xacml:2.0:profile:saml2.0:v2:schema:assertion:cd-01"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
               <urn2:Response xmlns:urn2="urn:oasis:names:tc:xacml:2.0:context:schema:os">
                  <urn2:Result>
                     <urn2:Decision>Permit</urn2:Decision>
                     <urn2:Status>
                        <urn2:StatusCode Value="urn:oasis:names:tc:xacml:1.0:status:ok"/>
                     </urn2:Status>
                  </urn2:Result>
               </urn2:Response>
            </urn1:Statement>
```

```
        </urn1:Assertion>
      </urn:Response>
   </soapenv:Body>
</soapenv:Envelope>
```

A4.3 Example SAML-XACML Request to AIPEP to Transfer Resource to a Remote Site

The SAML-XACML request below asks if a student is allowed to TRANSFER the resource with ID rid-123 to anywhere (location not specified in this example).

```
<soapenv:Envelope                         xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:urn="urn:oasis:names:tc:xacml:2.0:profile:saml2.0:v2:schema:protocol:cd-01"
xmlns:urn1="urn:oasis:names:tc:SAML:2.0:assertion"
xmlns:xd="http://www.w3.org/2000/09/xmldsig#"
xmlns:urn2="urn:oasis:names:tc:SAML:2.0:protocol"
xmlns:urn3="urn:oasis:names:tc:xacml:2.0:context:schema:os"
xmlns:urn4="urn:oasis:names:tc:xacml:2.0:policy:schema:os"
xmlns:urn5="urn:oasis:names:tc:xacml:2.0:profile:saml2.0:v2:schema:assertion:cd-01">
   <soapenv:Header/>
   <soapenv:Body>
<XACMLAuthzDecisionQuery
xmlns="urn:oasis:names:tc:xacml:2.0:profile:saml2.0:v2:schema:protocol:cd-01"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="urn:oasis:names:tc:xacml:2.0:profile:saml2.0:v2:schema:protocol:cd-01
file:/home/sfl/work/issrg/oasis-documents/xacml3/XACML-3.0-cd-1-updated-2009-May-
07/XSD/xacml-2.0-profile-saml2.0-v2-schema-protocol-cd-1.xsd"
    ID="A2009-10-13T12.57.07"
    Version="2.0"
    IssueInstant="2009-10-13T12:58:12.209Z">
<xacml-context:Request xmlns:xacml-context="urn:oasis:names:tc:xacml:2.0:context:schema:os">
    <Subject xmlns="urn:oasis:names:tc:xacml:2.0:context:schema:os">
       <Attribute                              AttributeId="urn:oid:1.2.826.0.1.3344810.1.1.14"
DataType="http://www.w3.org/2001/XMLSchema#string">
           <AttributeValue>
             student
           </AttributeValue>
       </Attribute>
    </Subject>
    <Resource xmlns="urn:oasis:names:tc:xacml:2.0:context:schema:os">
    <Attribute                    AttributeId="urn:oasis:names:tc:xacml:1.0:resource:resource-id"
DataType="http://www.w3.org/2001/XMLSchema#string">
      <AttributeValue>rid-123</AttributeValue>
    </Attribute>
  </Resource>
  <Action xmlns="urn:oasis:names:tc:xacml:2.0:context:schema:os">
    <Attribute                        AttributeId="urn:oasis:names:tc:xacml:1.0:action:action-id"
DataType="http://www.w3.org/2001/XMLSchema#string">
      <AttributeValue>TRANSFER</AttributeValue>
    </Attribute>
  </Action>
  <Environment xmlns="urn:oasis:names:tc:xacml:2.0:context:schema:os"/>
</xacml-context:Request>
</XACMLAuthzDecisionQuery>
  </soapenv:Body>
</soapenv:Envelope>
```

A4.4 AIPEP Response to Request in A4.3

The SAML response below indicates Success, and the XACML response indicates Permit. However the SAML response contains a StickyPoliciesCondition which requires the PEP to attach the encapsulated Sticky Policy to the outgoing transfer. Note that the outgoing sticky policy is identical to the incoming one in A4.1.

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">
  <soapenv:Body>
    <urn:Response                                  IssueInstant="2010-11-26T20:33:45.606Z"
ID="_8040772879247ba5a643b63c4645c0fc"        Version="2.0"        InResponseTo="A2009-10-
13T12.57.07" xmlns:urn="urn:oasis:names:tc:SAML:2.0:protocol">
      <urn:Status>
        <urn:StatusCode Value="urn:oasis:names:tc:SAML:2.0:status:Success"/>
      </urn:Status>
      <urn1:Assertion                              IssueInstant="2010-11-26T20:33:45.607Z"
ID="_e27a7bb87984db88d37a5102c642e296"                                         Version="2.0"
xmlns:urn1="urn:oasis:names:tc:SAML:2.0:assertion">
        <urn1:Conditions>
          <urn1:Condition>
            <tas3:StickyPoliciesCondition xmlns:tas3="tas3:to:be:decided:namespace">
              <sp:StickyPolicy        PolicyID="sticky-policy-1"        PolicyLanguage="PERMIS"
PolicyType="Authorization"                      TimeOfCreation="2010-08-09T01:00:00.000+01:00"
xmlns:sp="tas3:to:be:decided:namespace">
```

```
                <sp:PolicyAuthor                    Format="urn:oasis:names:tc:SAML:1.1:nameid-
format:X509SubjectName">cn=Stijn,ou=permisv5,c=gb</sp:PolicyAuthor>
                <sp:ResourceTypes>
                  <sp:ResourceType>personal:preferences</sp:ResourceType>
                </sp:ResourceTypes>
                <sp:PolicyContents>
                  <X.509_PMI_RBAC_Policy                                                 OID=""
xsi:noNamespaceSchemaLocation="file:/media/disk/home/sfl/issrg/svn/code/trunk/build/data/ope
npermis/bundle/issrg/pba/rbac/xmlpolicy/schemachecking/policy53.xsd"
xmlns="urn:oasis:names:tc:xacml:2.0:profile:saml2.0:v2:schema:protocol:cd-01"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
                    <SubjectPolicy>
                      <SubjectDomainSpec ID="everywhere">
                        <Include LDAPDN=""/>
                      </SubjectDomainSpec>
                    </SubjectPolicy>
                    <RoleHierarchyPolicy>
                      <RoleSpec OID="1.2.826.0.1.3344810.1.1.14" Type="permisRole">
                        <SupRole Value="UNSPECIFIED"/>
                      </RoleSpec>
                    </RoleHierarchyPolicy>
                    <SOAPolicy>
                      <SOASpec ID="anyone" LDAPDN=""/>
                    </SOAPolicy>
                    <RoleAssignmentPolicy>
                      <RoleAssignment>
                        <SubjectDomain ID="everywhere"/>
                        <RoleList>
                          <Role Type="permisRole"/>
                        </RoleList>
                        <Delegate Depth="0"/>
                        <SOA ID="anyone"/>
                        <Validity/>
                      </RoleAssignment>
                    </RoleAssignmentPolicy>
                    <TargetPolicy>
                      <TargetDomainSpec ID="anywhere">
                        <Include RegEx=".*"/>
                      </TargetDomainSpec>
                    </TargetPolicy>
                    <ActionPolicy>
                      <Action ID="TRANSFER" Name="TRANSFER"/>
                      <Action ID="SUBMIT" Name="SUBMIT"/>
                    </ActionPolicy>
                    <TargetAccessPolicy>
                      <TargetAccess>
                        <RoleList/>
                        <TargetList>
                          <Target>
                            <TargetDomain ID="anywhere"/>
                            <AllowedAction ID="SUBMIT"/>
                          </Target>
                        </TargetList>
                      </TargetAccess>
                      <TargetAccess>
                        <RoleList/>
                        <TargetList>
                          <Target>
                            <TargetDomain ID="anywhere"/>
                            <AllowedAction ID="TRANSFER"/>
                          </Target>
                        </TargetList>
                        <Obligations>
                          <Obligation                                              FulfillOn="Permit"
ObligationId="http://sec.cs.kent.ac.uk/obligations/AttachStickyPolicy"/>
                        </Obligations>
                      </TargetAccess>
                    </TargetAccessPolicy>
                  </X.509_PMI_RBAC_Policy>
                </sp:PolicyContents>
              </sp:StickyPolicy>
            </tas3:StickyPoliciesCondition>
          </urn1:Condition>
        </urn1:Conditions>
        <urn1:Statement                         xsi:type="urn:XACMLAuthzDecisionStatementType"
xmlns:urn="urn:oasis:names:tc:xacml:2.0:profile:saml2.0:v2:schema:assertion:cd-01"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
          <urn2:Response xmlns:urn2="urn:oasis:names:tc:xacml:2.0:context:schema:os">
            <urn2:Result>
              <urn2:Decision>Permit</urn2:Decision>
              <urn2:Status>
                <urn2:StatusCode Value="urn:oasis:names:tc:xacml:1.0:status:ok"/>
              </urn2:Status>
              <urn3:Obligations xmlns:urn3="urn:oasis:names:tc:xacml:2.0:policy:schema:os"/>
            </urn2:Result>
          </urn2:Response>
```

```
            </urn1:Statement>
          </urn1:Assertion>
        </urn:Response>
      </soapenv:Body>
</soapenv:Envelope>
```

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
3.0	7 December 2010	Version 3 released for internal review