Authorisation and Conflict Resolution for Hierarchical Domains

Giovanni Russello, Changyu Dong, Naranker Dulay
Department of Computing
Imperial College London
180 Queen's Gate, London, SW7 2AZ, UK
{g.russello, changyu.dong, n.dulay}@imperial.ac.uk

Abstract

In this paper we generalise the authorisation policy model supported by the Ponder policy language for hierarchically organised domains of managed objects to support subject-based policies and return policies. We describe the authorisation conflicts that can occur and present a strategy to automatically resolve them. In our model each action has four endpoints: the subject call, the subject return, the target call and the target return. Each endpoint can have an associated policy which is used to define constraints on which subjects are permitted to call which targets, and what is permitted to be transferred between subjects and targets. Subject-based policies aim to protect the subject from untrusted targets, while target-based policies aim to protect the target from unauthorised subjects. Subject-based policies are defined for and enforced by the subject's PEP, while target-based policies are defined for and enforced by the target's PEP. Although subject-based and target-based policies are separated, they can be uniformly specified in our framework.

I. Introduction

Current distributed systems involve a large number of applications which require an increasing variety of security mechanisms to fulfill their needs. The complexity of managing such systems results in high administrative costs and long deployment cycles. It becomes even worse as a system expands because the effort and time required for management becomes a burden. Therefore, it is desirable and becoming more and more critical that management procedures are automated to reduce administrative cost [1], [2]. For security it also desirable if management could come with a higher level of abstraction, so that the configuration details of security mechanisms and technologies used can be separated from the application logic.

Policy-based management is potentially an effective solution for the distribution, automation and dynamic adaptation of current and future systems. Policies are rules governing the choices in the behavior of a system [22]. In the context of policy-based management, security management can be defined as the support for specification of authorisation policies, and translation of these policies into information which can be used by security mechanisms to control accesses, manage key distribution, monitor and log security activities [21]. Security management should also support flexible adaptation mechanisms for responding to changing requirements, context, denial-of-service attacks, etc.

The contributions of this paper can be summarized as follows. First of all, a new model is described where authorisation policies can be uniformly specified and enforced for protecting both the subject and the target for a given action. Secondly, we describe a deterministic strategy for resolving authorisation policies conflicts on hierarchically organised domains of subjects and targets. Finally, our framework caters for neatly separating policy enforcement form the application logic.

Authorisation policies are mainly used for enforcing access control to check whether a subject is authorised to execute an action of a target. In this paper, we describe a framework based on the Ponder language [7], where authorisation policies can be uniformly specified for both the subject and the target of an action. With our framework, it becomes possible to specify and enforce authorisation policies to prevent the subject from performing actions that could be harmful for the subject or the subject's domain(s), e.g. preventing a web browser sending a request to a blacklist webserver. Furthermore, policies can be specified to prevent a subject from accepting a reply from an action that could threaten the integrity of the subject. In this case,

a policy can activate a filter to scrutinize the reply before it is passed to the subject.

Central to policy-managed systems is the resolution of conflicts that arise between policies. For instance, there might be two authorisation policies which permit and forbid the same action. Although previous work has investigated this problem in detail [13], no definitive solutions have been implemented for automatically resolving conflicts. In this paper, we provide a simple yet powerful strategy for conflict resolution that can deterministically provide a solution when a conflict arise.

This paper is organised as follows. In Section II, we provide an overview of related work. Section III briefly discusses the policy interpreter that we use in our framework and highlights the extensions introduced by our approach. Conflict resolution is described in Section IV. In Section V we discuss several approaches that could be used for implementing the policy enforcement mechanism. We conclude with Section VI providing future directions of our research.

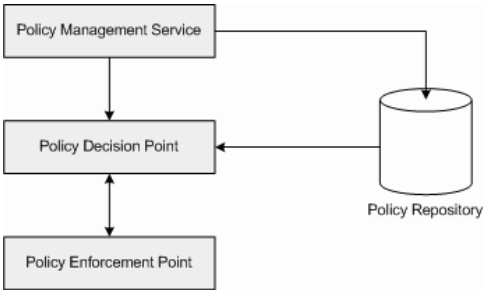

Fig. 1. The IETF policy architecture.

II. Background

A widely accepted architecture for policy-based management was proposed by IETF [25] as shown in Figure 1. The two main components in the architecture are the **Policy Decision Point** (PDP) and the **Policy Enforcement Point** (PEP). The PDP processes the policies, along with other data such as network state information, and takes policy decisions regarding what policies should be enforced and how this will happen. These policy decisions are sent as configuration data to the appropriate PEPes, which are responsible for installing and enforcing them. The other component proposed by the IETF is the **Policy Management Service** (PMS) which provides a user interface for specifying, editing, and administering policy. Having a language for security policy specification that provides a high abstraction level is crucial. In this direction, several research efforts have been conducted. In [3], the authors proposed a policy language for representing authorisation requirements. The language is a many-sorted first-order logic with a rule construct which is useful for stating structural properties of authorisation requirements. The language is declarative and has a semantics that is independent of implementation mechanisms and the semantics is efficiently computable which allowing efficient authorisation evaluation. The Authorisation Specification Language (ASL) [4], [5] is a logical language designed for an authorisation model which can support different types of policies within a single, unified system. For example, the users can specify closed policies in which all positive authorisations must be specified for some objects and open policies in which all negative authorisations have to be specified for the others. The language provides a general mechanism that is capable of implementing a number of different types of security policies, therefore making it possible to separate the implementations of policies from the implementations of access control mechanisms. Rei[12] is a policy framework designed for pervasive computing applications. Rei represents security and management policies in a semantic language like RDF-S, DAML+OIL or OWL. This allows different systems to share a model of policies, roles and other attributes. The language is not tied to any specific application and it permits domain specific information to be added without modification. In LGI [14], policies specify which actions the agent has to enforce upon the receipt or sending of messages. Policies use a simple Prolog notation. It assumes that policies are interpreted by trusted controllers at each agent's site. Ponder [7] is a declarative language that supports the specification of authorisation, obligation and other types of policies for managing distributed systems. Ponder uses an object-oriented approach which allows user to define different types of policies to meet specific security and administrative management goals.

The use of an effective language for policy specification makes it possible to separate the policy decision making from the application. As a consequence, the PDP can be generic and application-independent. Multiple applications can share the same PDP, therefore policies can be described uniformly across multiple domains. More importantly, it allows the design and implementation of security mechanisms to be separated from the application functionality. This means that application developers can focus more on the functionalities of the application without taking into consideration many of the security concerns. This approach is in line with the principle of Separation of Concerns [15].

Although it is possible to delegate security policy decisions to a universal policy engine, application developers still have to define in their application where the policy enforcement points are. Moreover, application developers are still responsible for correctly enforcing the decisions taken by policies which should be the concern of security

administrators. As pointed out by Filman, *et al.* in [8] not supporting a clear separation between the application functionality from system-wide properties, such as security, leads to an increase in the complexity of the system as a whole.

It seems that more abstraction for the PEP functionalities is needed. The functionalities of a PEP include checking the execution of the application and triggering the policy evaluation at appropriate times; collecting necessary information for the policy evaluation; enforcing the decisions from the PDP. If such functionalities are introduced at the application level then the level of dependency between the PEP and the application becomes too high. The PEP has to know which are the triggering points for the application. On the other hand, the application has to pass the required information for the policy evaluation and to enforce the decision taken by the PDP. If the PEP is such that the enforcement point could be abstracted from the specific application and the application is unaware of any details of policy enforcement then a complete decoupling would be achieved.

The work described in [6] is in line with this approach. Their work is an extension of the access control mechanism provided by Java's security framework [11]. It consists of several modules that have been introduced to map policies specified in Ponder [7] into Java security structures. Leveraging on the power and richness of the Ponder language, it becomes possible to specify more complex policies that can be analyzed for conflicts using an analysis tool provided by Ponder. The main advantage of this approach is that by using the security framework there is an effective decoupling of the PEP from application level. In fact, the points of policy enforcement are completely hidden from the application. On the other hand, because the Java security framework is limited to control resource access it cannot be used for intercepting other enforcement points.

Verhanneman, *et al.* [23] tried to achieve uniform application-level access control enforcement of organisation-wide policies by using Aspect-oriented Programming (AOP) [9]. They define *access interfaces* to specify the requirements an application needs to fulfill to enforce the policies. It specifies which requests are relevant to the enforcement of access control and what information is necessary to supply to take the control decision. The abstracted requirements are mapped to application-specific concepts by using a *view connector* at the application side. Each application needs its own view connector to bind it to an access interface. It uses Java Aspect Components [20] to implement a wrapper to intercept the method calls from the caller to callee so that view connectors can be configured at deployment time. The framework provides some flexibility but not so effectively. The implementations of security mechanisms and functionalities are not fully decoupled because developing view connectors is still the responsibility of the developers. Additionally, this framework focuses only on access control points for policy enforcement.

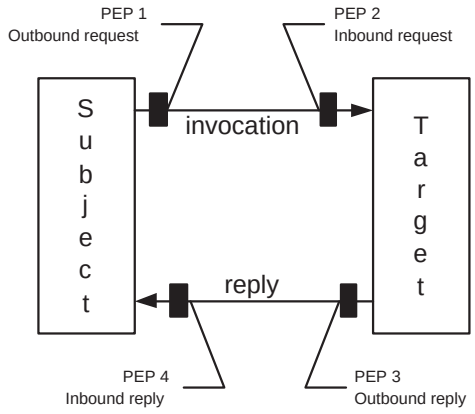

Fig. 2. Policy enforcement points.

To fill the gaps of the above approaches, we propose a framework where the PEPes can be specified for both the subject and the target. As shown in Figure 2, we specify four policy enforcement points:

- PEP 1: at this point, policies are enforced when the subject sends out a request to a target. PEP 1 policies are used to prevent a subject from calling an action on an untrusted target or filtering the data that is passed to the target (e.g. for privacy reasons). We name such policies *Subject authorisation (SA)* policies. Conditions can be defined on known properties of either the subject or the target or on contextual conditions, such as time of the day, location or cost of communication.
- PEP 2: is used for enforcing traditional authorisation policies for access control on the target. Policies are enforced when an action is received by the target. We name these policies *Target authorisation (TA)* policies.
- PEP 3: this point allows the target to apply policies after an action is executed but before the reply is sent back to the subject, i.e. to truncate or filter data that is sent back to the subject. Just denying the subject the right to perform the operation is not sufficient to cover this case. We name these policies *Target-return authorisation (TRA)* policies.
- PEP 4: is used for enforcing policies when the subject receives the reply. PEP 4 policies can be used to protect the integrity of the subject from malicious or buggy data sent from the target. We name such policies *Subject-return authorisation (SRA)* policies.

In object-oriented systems we treat subjects and targets as objects, and actions as methods. Intercepting method invocation or reply can provide the base information

for policy evaluation. In addition, the parameters of the method call and the return value of the reply can provide more information if needed.

In the following section, we provide more details about our framework and present the syntax of authorisation policies.

III. The Policy Interpreter

The policy interpreter that we extended implements the Ponder2 language being developed at Imperial College [16]. The interpreter supports obligation policies (event-condition-actions) and target-authorisation policies written in XML. The interpreter organises the entities on which policies operate in hierarchical domains of *managed objects*. A managed object has a management interface that the object has to implement in order to be managed by the interpreter. Domains allow the classification and grouping of managed objects in a hierarchy. Furthermore, domain paths can be used to address subjects and targets in policy specifications. Before a managed object is added to one or more local domains, the policy interpreter authenticates the managed object.

Different nodes of a distributed system have their own policy interpreter which maintains its own local domain hierarchy. Policies refer to local domain paths and are enforced locally. When a managed object wants to execute an action on a target object that is located on a remote node proxies are created by each interpreter and added in the local domains.

Let us consider the scenario shown in Figure 3 where a nurse ($n1$) working in $Hospital1$ wants to get the medical record of a patient ($p1$). However, the record is not in her hospital but is stored in $Hospital2$.

The following steps are executed by the policy interpreters of each hospital. When the nurse makes the request for accessing the patient record (1), the Naming service of the interpreter at $Hospital1$ resolves (transparently to he nurse application) the patient application's URL, contacting the Naming service at $Hospital2$. The communication between the Naming services of each host is handled by the respective Comm service (2). The Naming service at $Hospital2$ forwards the request to its local Authentication service (3). The Authentication services of each hospital conduct a negotiation for the credentials of the nurse (4). Once the nurse is authenticated, the Authentication service at $Hospital2$ creates a nurse proxy and inserts it in its local domain structure (5). However, the nurse proxy is inserted in the $nurse$ domain contained in the $external$ domain. In this way, it is possible to differentiate between nurses working in the hospital and external ones and more restrictive policies can be defined for external nurses. Once the nurse is authenticated, the Naming service at $Hospital2$ retrieves the reference for the patient $p1$ (6) and returns it to the Naming service at $Hospital1$ via the Comm service (7). The obtained patient reference is passed to the Authentication service (8) at $Hospital1$ and authenticated (4). After authentication, a proxy[1] for the remote patient object is created and inserted in the local domain structure (9). Now the nurse $n1$ makes the invocation (10) via the local proxy of the patient. This request is forwarded to $Hospital2$ (11-12) using the Comm service, and it is executed as a local operation by the nurse proxy (13-14) on $Hospital2$.

The evaluation and enforcement of authorization policies is executed during step (10) for PEP1 and PEP4, and during step (14) for PEP2 and PEP3. This simple approach allows us to specify and enforce authorization policies in a completely decentralized way. Furthermore, the negotiation between authentication services is currently based on credentials, such as certificates that assert the target and subject roles. However, we foresee, as discussed in our future work (see Section VI), the use of a more sophisticated authentication service based on *automated trust negotiation* [17], [18], [19].

In the following subsection, we outline the basics of our authorisation policies.

A. Authorisation Policies

In the domain structure maintained by the policy interpreter, a domain can only be reached using one single path. In other words, a domain can be the child of only one other domain. However, instances of managed objects are allowed to be present in more than one domain. In this way, if a domain represents a role then an instance of a managed object can be associated with multiple roles.

It is quite common in a policy-managed system to adopt a default authorisation policy: every action is allowed or every action is forbidden. In our framework we allow the policy administrator to specify the default authorisation policy. This is done by setting the value of the attribute **`defaultAccess`**. The possible values are:

- **`ALL+`** indicates that by default all actions are allowed. To restrict accesses negative authorisation policies must be defined.
- **`ALL-`** indicates that by default no action can be executed unless a positive authorisation policy is defined.

Once the value of this attribute is assigned to a domain structure it cannot be changed. Changing the value of this attribute after several authorisation policies have been specified could alter the meaning of those policies.

[1] A proxy for the nurse object is also created at $Hospital2$.

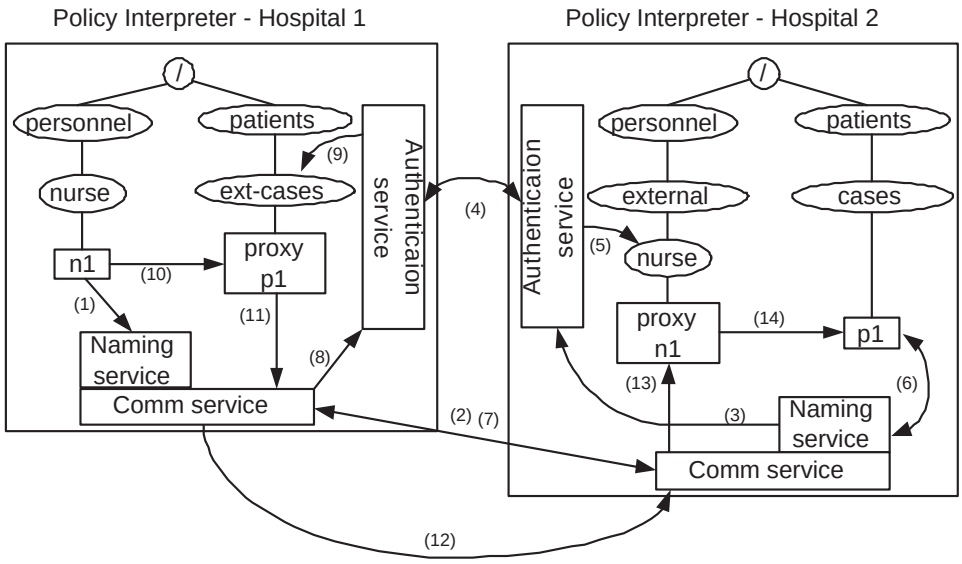

Fig. 3. A remote invocation scenario.

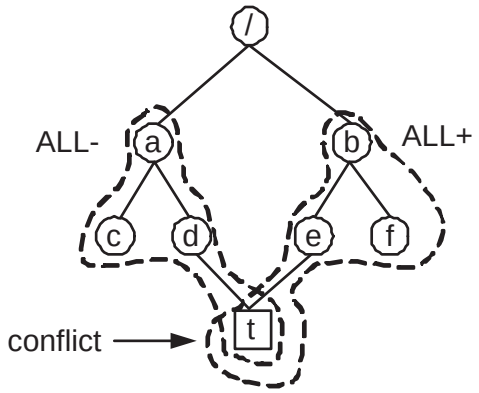

Fig. 4. Conflicts for default properties if defined per domain.

Ideally, this attribute could be defined on a domain basis. The subdomains and instances contained in a given domain inherit the values of the attribute. However, because the same instance of a managed object can be contained in multiple domains it could be the case that the values of the **defaultAccess** attribute could conflict. Figure 4 depicts the case in which an instance t is contained in two domains, d and e. The **defaultAccess** attribute value for domain d conflicts with the **defaultAccess** attribute value of domain e. In fact, domain d is a domain sub-structure where by default no actions are allowed. On the other hand, domain e is part of a sub-structure where all actions are allowed and negative policies override positive ones. To avoid this type of conflict, the **defaultAccess** attribute is required and only permitted for the root domain and its value propagates down to the domain structure to all the subdomains and contained instances.

B. Policy Specifications

In this subsection, we provide examples of policies that it is possible to specify using our approach. The policies that we consider are for mobile healthcare agents. We focus on subject-enforced policies.

Policy 1: shows a negative subject authorisation (SA-) that prevents a mobile patient agent requesting treatment from an untrusted medical service e.g. a medical service that cannot provide a valid certificate signed by the National Health Service (NHS).

```
subject
   auth- patientAgent.requestTreat()→MedService
      when !certified(MedService, NHS)
```

Policy 2: shows a negative subject return authorisation (SRA-) applied on the patient agent. The policy denies to the agent access to the treatment returned by the medical service when the returned treatment is signed by a GP that is not recognized by the NHS. Note: to ensure call-return pairing SRA policies must be appended to positive SA policies. Similarly TRA policies must be appended to positive TA policies.

```
subject
   auth+ patientAgent.requestTreat()→MedService
      when request.condition=SERIOUS
   return-
      when !certified(reply.GPSignature, NHS)
```

Positive authorisation policies can be used to filter the data that is supplied or returned. The filter is specified by using `filter` in the action clause. Filtering policies must be positive authorisations because no transformation needs to be applied if the action is forbidden.

Policy 3: shows a positive subject authorisation (SA+) for an employee. The employee has to provide to the GP of the company where the employee works her medical record. For privacy reasons, the employee wants her psychiatric data removed from her medical record. The policy applies a filter that nullifies the sensitive field from the record. The filter is executed by the subject's policy interpreter by intercepting the call, nullifying the psych field and then forwarding the call to the target.

```
subject
   auth+ employeeAgent.rec()→CompanyGP
      filter request.psych:=NULL
```

Policy 4: shows a traditional positive target authorisation (TA+) policy for a nurse agent that has to perform accesses on the patients' medical records in a hospital. According to this policy, a nurse agent can access the medical records of a patient when the nurse is on duty on the ward where the patient is being treated.

```
target
   auth+ nurseAgent→medicalRecordDB.readrec()
      when ward(nurseAgent)=ward(request.patient)
```

Policy 5: shows a positive target return authorisation policy (TRA+) for an insured patient that modifies a terminal diagnosis asking that the patient contact the Medical Service.

```
target
   auth+ patientAgent→MedService.requestTreat()
      when insured(request.InsuranceNo)
   return+
      when reply.diagnosis=TERMINAL
      filter reply.diagnosis:=CONTACT_US
```

It should be noted that the framework realizes a complete separation of concerns. In fact, all the details about checking the credentials of the target, the target's reply, and the application of filters on sensitive data are specified outside the logic of the application. These details are isolated and captured in the policy specification.

IV. Conflict Resolution

When dealing with policy based systems, it is unavoidable that conflicts arise in the set of policies. Ideally conflicts are detected by static analysis of the policy set. However it is often not possible to perform such analysis on policies that depend on run-time state. This issue is more acute in the case of large systems where the policies are specified as the system evolves. In our model, we define a conflict-resolution strategy that is used statically and dynamically to determine which policy takes precedence. The strategy aims to provide policy administrators with a range of desirable policy precedence behaviours. Typically, conflicts arise when multiple policies apply to the same (subject, target, action)-triple. Therefore, it is necessary to provide rules to define the precedence between conflicting policies. In this paper, we focus on modality conflicts, that is inconsistencies that arises when more than one policy with modality of opposite sign apply to the same subject, target and action.

First of all, in our model there are two default modalities that administrators can choose. The first modality allows all actions to be performed unless an authorisation policy is specified. In this case, the restrictions imposed by the authorisation policy must be satisfied to complete the action. The second modality is more restrictive inasmuch as all the actions are prohibited unless an authorisation policy is specified. In any event, whatever the default modality is, modality conflicts can arise.

To determine the precedence between two or more policies we based our conflict resolution algorithm on domain nesting. The domain nesting resolution gives precedence to policies that apply to a more specific instance of subjects, targets, or both. In other words, a policy that applies to a subdomain is more specific than a policy that applies to any ancestor domains. The main strength of this approach is that it is intuitively applicable to a domain-based system. For instance in Figure 5-(a) policy $p2$ takes precedence over policy $p1$ being $p2$ more specific than $p1$.

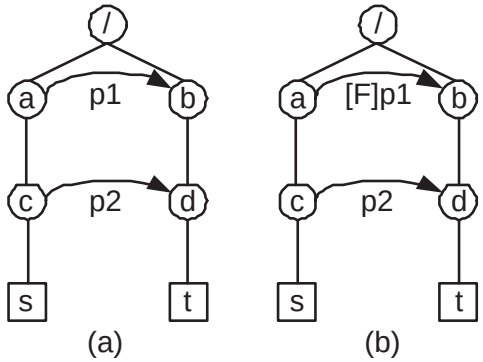

Fig. 5. Examples of priority based on domain nesting (a) and final status (b).

However, this principle does not apply to all situations. Sometimes it is desirable that a global policy overrides more specific ones. For supporting these cases, in our framework it is possible to use *special* global policies that override any specific policies defined in the subdomain structure. To define such a policy the keyword **`final`** must be used in the definition of the policy. Final authorisation policies can only be defined on domains, since it does not

make sense to define such policies on object instances. Figure 5-(b) shows a scenario where the final policy $p1$ overrides policy $p2$.

There are cases in which there is no way of determining a precedence based on domain nesting and final policies. This is the case when the same managed object resides in different domains and conflicting policies are defined on those domains. For such cases, in our framework negative authorisation overrides positive authorisation.

A. Conflict Resolution Strategy

The strategy for conflict resolution is based on the following preliminary steps:

1) First search for the uppermost final policy that is applicable on the given (subject, target, action)-triple.
2) If a modal conflict arises between applicable final policies of the same domain level then any negative authorisation gets priority.
3) If no final policy is found then search for the most specific policies that are applicable to the triple.
4) If a modal conflict arises then give higher priority to the negative one.

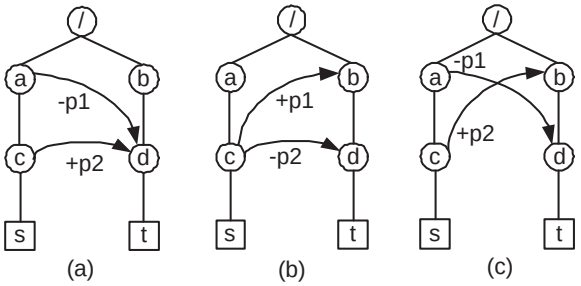

Fig. 6. Examples where the specificity of a policy cannot be determined.

However, it is possible that policies could be specified on subjects that are at different levels in the domain structure but on the same target, and vice-versa. Figure 6 depicts several such cases. Case 6-(a) is when an action is authorised for a restricted group of entities. For instance, in general, nurses are not allowed to access a patients' records (negative policy $p1$), but only if a nurse is on duty on the same ward of the patient (positive policy $p2$). Case 6-(b) captures the case when an action on a specific domain of entities must not be allowed (negative policy $p2$), but the same action is permitted for a more general set of targets (policy $p1$). For instance, a medical researcher is allowed to access all the patients' records expect for certain cases. And case 6-(c) captures the case in which an action on a specific domain of targets must not be allowed to a general domain of subjects. However, a more specific domain of subjects are allowed to execute the same action on a general domain of targets. For instance, in general, nurses cannot access the patients' personal information, contained in domain d. However, nurses in Emergency Room can have access to medical records (contained in domain b) and in particular to the personal information of patients. The selection of an eligible policy, in such cases, is done using the length of the path from the subject to the target using the policy arc. Two cases can be distinguished: resolving for final policies and resolving for normal non-final policies.

For the first case, the policy with the longest path is selected. For instance, in Figure 6-(a) if $p1$ and $p2$ are both final policies, then policy $p1$ will override $p2$. In fact, the longest path from subject s to target t is through policy $p1$ ("$s, c, a, p1, d, t$" compared to "$s, c, p2, d, t$") . In the case of Figure 6-(b) again the longest path is through policy $p1$ ("$s, c, p1, b, d, t$" compared to "$s, c, p2, d, t$"). The case in Figure 6-(c) is more interesting since both paths have the same length. For a case like this, we give higher importance to the path of the subject[2]. Therefore, the policy $p1$ overrides policy $p2$ because policy $p1$ is more general than $p2$.

For the second case, when resolving normal non-final policies, the policy that provides the shortest path is selected. Consequently, in Figure 6-(a) and (b) policy $p2$ overrides policy $p1$. For the case in Figure 6-(c), policy $p2$ overrides $p1$ because $p2$ is more specific than $p1$ with respect to the subject path.

When the subject or the target of an action are in multiple domains, then for each domain path we search for an eligible policy using the same strategy as shown before. All the eligible policies found for each path are evaluated according to the negative precedence rule. In other words, if one negative policy is found in any path then this overrides all the other policies and the action is not allowed. The rationale behind this decision is to select and enforce the policy that does not affect the integrity of the system. If all the policies are positive then the action is allowed. Otherwise, if no eligible policies have been found then the policy specified by the **`defaultAccess`** attribute is enforced.

Let us consider the examples shown in Figure 7 to see in practice how conflicts are resolved. In Figure 7-(a), the subject is contained in two domains, c and e. The conflict resolution strategy is used for path "$s, e, p1, d, t$" and the negative policy $p1$ is found. For the path "$s, c, p2, d, t$" the

[2]The reason for this strategy is that users are familiar with the concept of specifying policies to protect their assets (targets) from being accessed by other entities (subjects). In the next future, we foresee a new release of our framework with a more complex strategy that discerns between the specificity of subject and target paths depending on whether the policy that should be enforced is of target or subject type, respectively.

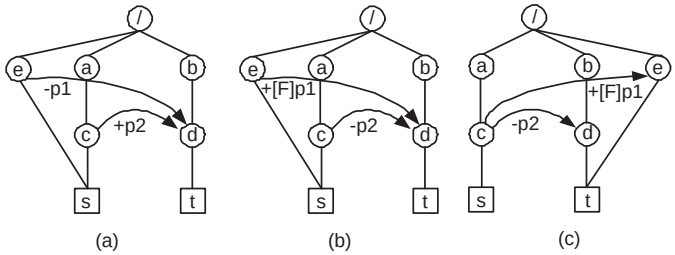

Fig. 7. Examples where the subject and the target are contained in more than one domain.

positive policy $p2$ is found. Because of the negative priority rule, policy $p1$ is selected. It could seem that policy $p2$ should be applied over $p1$ using the domain nesting rule, but since $p1$ and $p2$ are defined on different paths, the rule cannot be applied.

In Figure 7-(b) and (c) a final policy is defined in one of the paths where the subject or the target are contained. For case 7-(b) the search is started from the subject. In path "$s, e, p1, d, t$" the final positive policy $p1$ is found and in path "$s, c, p2, d, t$" the negative policy $p2$ is found. Again here the negative precedence rule is applied and policy $p2$ overrides policy $p1$. The rationale behind this is that the final status of policy $p1$ makes sense only on the path where the policy is defined. For different paths, it is difficult to determine the specificity of a policy because the domain nesting for different paths is not comparable. This means that when compared to policies defined on different paths, final policies are treated as normal ones.

Figure 7-(c) shows the case in which a target is contained in multiple domains. In this case, for path "$s, c, p1, e, t$" the final policy $p1$ is found and for path "$s, c, p2, d, t$" the negative policy $p2$ is found. Again here, policy $p1$ loses its status of final and the policy $p2$ overrides it for the negative priority rule.

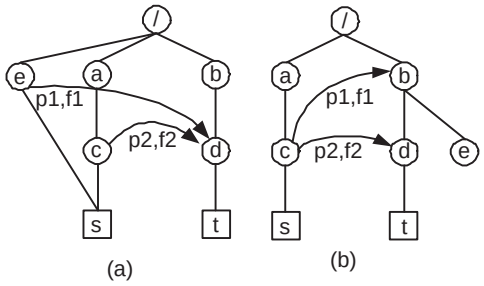

Fig. 8. Examples where multiple filters can be applied.

B. On Filtering

The use of filters in positive authorisation policies brings up some interesting situations that should be discussed. For instance, Figure 8-(a) shows the case in which two filtering policies could be applied to a subject that is contained in two different domains. Policy $p1$ applies filter $f1$ to the reply and policy $p2$ applies a different filter $f2$. Here, it is not about solving a conflict, because both policies are positive. The problem is how to apply a filter that returns the appropriate information as the policy administrator wanted to express with these policies.

A case could be that a doctor is also a researcher in the hospital and he wants to access patients' records, specified as follows: $rec :< name, age, address, patology >$. Policy $p1$ allows the doctor to access the record of a patient but filter $f1$ removes the address of the patient ($f1 \rightarrow rec.address = NULL$). On the other hand, policy $p2$ allows a researcher to access the patient record but the filter $f2$ removes the patient's name ($f2 \rightarrow rec.name = NULL$). For cases like this one, the intersection of the filters's fields is applied. For the specific example of the doctor/researcher it means that no filtering is applied.

It could be the case that for the same attribute different filtered values are specified. For instance, filter $f1$ specifies that the value of attribute x is substituted with value v' while filter $f2$ specifies that for the same attribute x the value to substitute is v''. To securely solve filtering conflicts, the value of the attribute x is nullified. Although this solution changes the semantic of the filters, we reduce the risk that meaningful information is leaked out. When such conflicts arise, the policy interpreter, logs a warning to alert the policy administrator of the possible problem.

Another possible problem is presented in Figure 8-(b). In this case, different filters can be applied to the same target and therefore their union should be applied to the result. For instance, filter $f1$ of policy $p1$ could nullify fields that are common to all the instances that are in domain b. On the other hand, filter $f2$ removes fields specific of instances contained in domain d. Returning to the hospital example, domain b contains patients and domain d contain terminal patients. Then, filter $f1$ removes the identity and filter $f2$ removes the life expectancy of a terminal patient. Applying the union of the filters removes all the necessary information that needs to be protected by both policies.

V. Policy Enforcement Mechanisms

Crucial to our approach is the realization of a PEP mechanism such that (i) it supports a fine-grained level of enforcement point specification and (ii) it is completely transparent to the application logic.

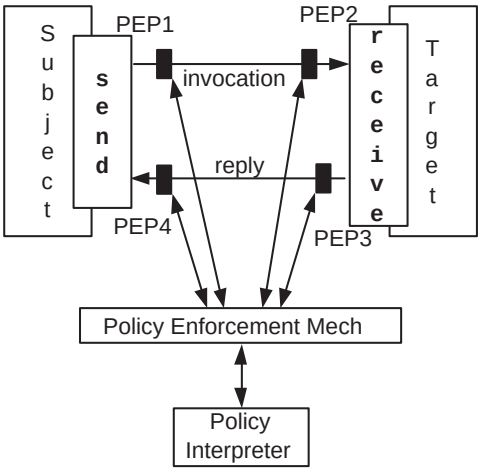

Fig. 9. The modules implemented in our framework to provide a complete control over authorisation.

The realization of a PEP could be achieved using the standard Wrapper Pattern. The idea is that any managed component code is *wrapped* by a piece of code that intercepts all the inbound and outbound calls to and from the component (actually this is our managed object interface). Each time a call is intercepted, the wrapper passes the necessary information to the Policy Interpreter taking policy decisions. The clear advantage of this method is that we have complete control over the wrapper design and we can easily customise it for our needs.

A Java-based solution that supports the wrapper pattern is the Java Management Extension (JMX) [10]. JMX provides application developers with a powerful yet simple solution for managing resources. In JMX resources may be objects, devices, and services. A resource can be enabled to be managed through the JMX framework if it is enabled via a Managed Bean (MBean). An MBean is a special Java bean that exposes via a standardized interface (defined by the JMX specification) attributes and methods of the resource that it manages. MBeans can be enabled to emit notifications when certain events happen. Given the fine-grained instrumentation that can be achieved with a properly specified MBean interface and its notification capability, JMX could be used for implementing our PEPs and provide access to the management capabilities supported by JMX.

With the release of the J2SE 5.0 the Java Virtual Machine Tool Interface (JVMTI) was introduced to provide an interface that software agents can use to control and monitor Java applications. Agents use the functionality exposed by the interface to be notified when events occur in the application, and to query and control the application during execution. The agent is passed as an argument when the JVM is started. Among the events that an agent can intercept there are those that capture when the execution enters and exits a method. JVMTI allows the agent to retrieve information regarding method call, such as object type of the caller and the callee, the parameter values passed in the method invocation, and the value that the method returns.

An alternative approach would be to use AOP, as done in [23]. This technique requires that the application code is modified with the injection of aspect-specific code. Crucial in AOP is the specification of where the aspect code must be injected. In Java Aspect Components (JAC), this is done transparently to the application. However, this requires that descriptor files must be supplied to the JAC middleware where those points are specified. In particular for the approach described in [23], application developers have to specify descriptors that must be customized for the specific security policy that is going to be enforced. For instance, if the policy requires the values of some parameters, the application developer has to specify in the descriptor how such parameters are to be retrieved.

It should be noted that the design of our framework is independent of the actual mechanism used for implementing a PEP. As shown in Figure 9, for the implementation of the Policy Enforcement Mechanism all the above approaches could be used. This has the advantage of allowing our framework to enforce policies across systems implemented using different technologies.

VI. Conclusions and Future Work

In this paper we presented a model for the specification and enforcement of authorisation policies defined over hierarchically organised domains. The contributions of this paper are the followings. Firstly, in our model authorisation policies can be used to protect both the subject and the target of an action. It also supports the validation and filtering of information flows between subjects and targets. This fills the gap of previous approaches where policies could be specified and enforced on the target side only. Secondly, the model includes a strategy that deterministically resolves conflicts that could arise between authorisation policies while minimising leaks due to filtering conflicts. Finally, our framework treats policy specification and enforcement in line with the SoC principle. As a matter of facts, the application logic is completely agnostic of authorisation policies and how/when they are enforced at run-time.

There are several areas that we want to investigate as future research. First of all, we want to extend the policy interpreter to include static checks for authorisation policy conflicts, for example, when policies are loaded or

unloaded. Static checks can provide an early indication of conflicts and can also enable optimisations that speed up policy enforcement. Secondly, we want to investigate how dynamic changes of the domain hierarchy impact policy enforcement and conflict solving. The main objective here is to investigate how to maximise concurrency and minimise locking of data-structures in the policy interpreter. Finally, we are looking at integrating a trust model, policies and mechanisms. Since the action interception mechanism is independent from the actual authorisation model, we can easily integrate a trust component that can take decisions on whether given subjects, targets, actions are 'trustworthy'. The component would recommend trust-levels or even authorisation policies. These levels or policies can change over time, providing a very flexible framework in comparison with the yes-or-no approach of classical security models.

Acknowledgments

This research was supported by the UK's EPSRC research grant EP/C537181/1 and forms part of the CareGrid, a collaborative project with the University of Cambridge. The authors would like to thank the members of the Policy Research Group at Imperial College for their support.

References

[1] M. Feridun, M. Leib, M.H. Nodine, J.C. Ong, "ANM: Automated network management system." *Network, IEEE , vol.2, no.2*, pp.13-19, Mar 1988

[2] John Strassner, "*Policy-based network management, solution for the next generation.*" Morgan and Kaufmann, 2004

[3] Thomas Y. C. Woo and Simon S. Lam, "authorisations in Distributed Systems: A New Approach" *Journal of Computer Security , vol.2, no.2-3*, pp.107-136, 1993

[4] Sushil Jajodia, Pierangela Samarati, V. S. Subrahmanian, and Elisa Bertino. "A Unified Framework for Enforcing Multiple Access Control Policies." *In Proc. ACM SIGMOD International Conference on Management of Data 1997*, pp. 474 – 485, 1997.

[5] Sushil Jajodia, Pierangela Samarati, and V. S. Subrahmanian. "A Logical Language for Expressing authorisations." *In Proc. 1997 IEEE Symposium on Security and Privacy*, 1997.

[6] Corradi et al. A flexible access control service for Java mobile code. In 16th Annual Computer Security Application Conference (ACSAC'00)s.

[7] N. Damianou, N. Dulay, E. Lupu and M. Sloman. "The Ponder Policy Specification Language." *In Proc. 2nd IEEE International Workshop on Policies for Distributed Systems and Networks*, pp. 18–38, 2001.

[8] R. Filman, S. Barrettt, D. Lee, and T. Linden. "*Inserting Ilities by Controlling Communications*." In *Communications of the ACM*, 45(1):116–122, Jan. 2002.

[9] R. E. Filman and D. P. Friedman. "*Aspect-Oriented Programming is Quantification and Obliviousness*." Workshop on Advanced Separation of Concerns, OOPSLA, October 2000.

[10] Java Management Extension Specifications. http://jcp.org/aboutJava/communityprocess/final/jsr003/index3.html

[11] Java Security White Paper. http://java.sun.com/developer/technicalArticles/Security/whitepaper/JS_White_Paper.pdf

[12] L. Kagal, T. Finin and A. Joshi. "A policy language for a pervasive computing environment." *In Proc. 4th IEEE International Workshop on Policies for Distributed Systems and Networks*, pp. 63–74, 2003.

[13] E. Lupu and M. Sloman. "Conflicts in Policy-Based Distributed Systems Management." *IEEE Transaction on Software Engineering*, pp. 852–869, Vol. 25, No. 6, 1999.

[14] N. H. Minsky and P. Pal. "Law-Governed Regularities in Object Systems - Part 2: A Concrete Implementation." *Theory and Practice of Object Systems (TAPOS)*, John Wiley. 2, 1997.

[15] D. L. Parnas. "On the criteria to be used in decomposing systems into modules." *Communications of the ACM*, 15(12):1053-1058, December 1972.

[16] The Ponder2 Project http://ponder2.net/

[17] W. H. Winsborough, K. E. Seamons, and V. E. Jones, "Automated trust negotiation." in *DARPA Information Survivability Conference and Exposition, 2000. DISCEX '00*, pp. 88–102 vol.1, 2000.

[18] J. Li, N. Li, and W. H. Winsborough, "Automated trust negotiation using cryptographic credentials." in *CCS '05: Proceedings of the 12th ACM conference on Computer and communications security*, pp. 46–57, 2005.

[19] C. Dong, and N. Dulay, "Privacy Preserving Trust Negotiation for Pervasive Healthcare." in *Proceedings of the 1st International Conference on Pervasive Computing Technologies for Healthcare*, 2006.

[20] R. Pawlak, L. Seinturier, L. Duchien, and G. Florin. "*JAC: A Flexible Framework for AOP in Java*." In *Reflection'01*, volume 2192 of Lecture Notes in Computer Science, pages 1–24. Springer-Verlag, September 2001.

[21] M. Sloman, J. Magee and K. Twidle and J. Kramer. "An Architecture for Managing Distributed Systems." *In Proc. 4th IEEE Workshop on Future Trends of Distributed Computing Systems*, pp. 40–46, 1993.

[22] M. Sloman and E. Lupu. "Security and Management Policy Specification." *IEEE Network*, pp.10–19, Vol. 16, Issue 2, March, 2002.

[23] T. Verhanneman, F. Piessens, B. D. Win and Wouter Joosen "Uniform Application-level Access Control Enforcement of organisation-wide Policies." *In Proc. 21st Annual Computer Security Applications Conference*, pp. 431–440, 2005.

[24] A. Uszok, J. Bradshaw, R. Jeffers, N. Suri, P. Hayes, M. Breedy, L. Bunch, M. Johnson, S. Kulkarni and J. Lott. "KAoS policy and domain services: toward a description-logic approach to policy representation, deconfliction, and enforcement." *In Proc. 4th IEEE International Workshop on Policies for Distributed Systems and Networks*, pp. 93–96, 2003.

[25] R. Yavatkar, D. Pendarakis and R. Guerin. "A Framework for Policy Based Admission Control." *IETF RFC 2753*, 2000.