APOLLON: TOWARDS A SEMANTICALLY EXTENSIBLE POLICY FRAMEWORK

Julian Schütte
Fraunhofer AISEC
Garching (near Munich), Germany

Keywords: Semantic Policies, Policy Framework, Middleware Architecture

Abstract: With increasing complexity of distributed systems, the ability to control access rights and security settings becomes more and more critical. Especially pervasive systems with ad hoc connectivity and semantic discovery of services are a challenging environment when it comes to managing their security. Most policy frameworks come with a pre-defined policy model whose expressiveness can usually not be extended and is thus not adaptable to a high-level security model as it might be defined by a company or a specific application. In order to overcome these limitations we designed Apollon, a policy framework featuring a modular policy model which can be extended or reduced as required by an application. While we describe an exemplary policy model represented in Description Logics (DL), Apollon uses in general a hybrid decision engine which allows to combine DL-based reasoning with dedicated decision plug-ins. In this paper, we present the software architecture of Apollon, introduce a basic attribute-based policy model and show by the example of a dynamic role-based access control model how its expressiveness can be extended.

1 INTRODUCTION

Context-aware and “intelligent” environments are on the rise, promoted by the advent of powerful and mobile devices like smartphones. These systems are characterized by a distributed software architecture and heterogeneous devices which join the network in an ad hoc fashion and are discovered and connected at run time. Lately, a number of middleware systems which facilitate the creation of pervasive-systems-based applications has been developed. While more and more such applications occur, controlling access rights and security settings throughout the whole application is still an open issue. Some of the main challenges in that area are:

Most frameworks feature a specific policy model which may or may not be suited for a system. As the applications in a distributed system evolve, additional demands on the security model might arise and in case the policy framework does not support them, re-deployment of a new suited framework or other workarounds become necessary, which is obviously not optimal. Furthermore, it is critical that the specification of policies is made as easy as possible. To date, policy specification is usually an error-prone task and the likelihood that the effective policy does not fully match the developer’s original intention is high. With the increased complexity of pervasive systems, it is all the more important, that putting a high-level security model into a set of rules is facilitated as far as possible.

Along with that goes the demand for the ability to express high-level security models. Many policy languages feature low-level and fine-granular rules. The problem with that is that the actual security model becomes buried in a huge set of complex rules which mix up the high-level security model with domain knowledge, which makes it on the one hand very hard to recognize the actual security model from the policy and requires on the other hand that policies need to be modified whenever the domain knowledge has changed. Therefore, it must be possible to specify policies at a more abstract level which is closer to the actual security model and easier to understand for developers. These policies must then automatically be mapped into concrete enforceable actions at run time.
Finally, verifiability of security properties should be supported. Whenever the policy needs to be modified, developers must be able to verify that the security model is still effective as intended. For example, it should be possible to verify that a certain constraint, such as separation of duty, is still in effect after adding further rules to the policy.

In this paper we present Apollon, a policy framework which addresses the aforementioned challenges. It provides an extensible software architecture which allows to integrate various access control and obligation models whose expressiveness can be increased by means of plug-ins according to the needs of an application. As an exemplary policy model, we describe a dynamic role based access control (DRBAC) model which is fully expressed in Description Logics (DL) so that a formalization of the policies used is inherently provided, and allows verification and analysis of policies. As Apollon is supposed to be used in pervasive systems, we further address typical issues like the resolution of policy conflicts between different policy domains as well as the negotiation of security mechanisms in a group of peers. In this paper, however, we will mainly concentrate on the basic design of Apollon and the exemplary RBAC model and only sketch how such advanced features can be realized.

The rest of the paper is structured as follows: in section 2 we review the current state of the art. In section 3, we introduce the Apollon framework and describe the policy decision engine, as well as the concept of using semantic knowledge bases to represent security policies. An exemplary basic authorization model, based on DL is then introduced in section 6 and examples of further high-level security models on top are given. In section 6, we then discuss the prototype implementation of Apollon and point out its strengths and weaknesses. Section 7 finally concludes the paper and outlines our future work.

2 Related Work

XACML (OASIS, 2005b), the probably commercially most used access control policy language, aims at controlling access to web services and features an attribute-based access control model. It is not limited to web service infrastructures but rather aims at providing an extensible language which can be adapted to different types of applications. However, there are some shortcomings which make XACML not the optimal policy languages for pervasive systems: as the name suggests, XACML focuses only on access control and is based on the COPS (Boyle et al., 2000) policy model which suggests to evaluate access requests from a subject to a resource in a synchronous way (i.e. a policy decision is only triggered by an access request which is stalled during the decision process). Further, it requires subjects and resources to be specified at design time by means of static attributes, which is problematic in pervasive systems where services are discovered and invoked at run time without a priori knowing their exact identity. Other authors have further criticized that XACML is not based on a consistent formalism and have undertaken attempts of formalising XACML (Kolovski et al., 2007; Bryans, 2005; Kolovski, 2008; Kolovski, 2008; Humenn, 2003). To the best of our knowledge, none of them covers the full spectrum of the specification so that a formal verification of XACML policies is not possible.

Further, many policy frameworks targeting distributed systems have been proposed: Ponder (Pon, 2008; Twidle et al., 2009) is a policy framework which has explicitly been designed for pervasive systems and makes use of a proprietary policy language. Besides access control it also features asynchronous event-condition-action (ECA) policies. Though the Ponder policy language is not based on a formal representation, the authors have shown in (Lymberopoulos et al., 2004) that validation of policy properties in Ponder is possible. Cassandra (Becker and Sewell, 2004) is a framework specifying distributed role-based access control using a Datalog-based language whose expressiveness can be extended by the choice of a constraint domain. The framework is similar to our approach in that its expressiveness (and with that, complexity) can be adapted to the needs of an application. It differs however from our approach in that it focusses on RBAC only (e.g., does not support asynchronously triggered obligation policies) and does not consider semantically modelling domain knowledge separate from the actual security policy. Combining semantics and policies has been proposed by a number of authors before: for instance, the policy framework Rein (Toninelli et al., 2005; Tonti et al., 2003; Kagel et al., 2006; Lalana Kagel, 2006), uses description logic to model and decide access control policies. Rein supports an extensive policy model including advanced concepts like delegation and obligations. Similarly, KaoS (Uszok and Bradshaw, 2004; Uszok et al., 2003) relies on policies modelled in OWL (Motik et al., 2009) and extends its expressiveness by integration of the Java Theorem Prover. In contrast to our proposal, both frameworks pre-define a policy model which cannot be adapted to different needs. Another framework which uses description logic to model policies and contexts is Proteus (Toninelli et al., 2007), which supports access control as well as obligations. The author’s goal is
to achieve self-adaptation by inferring ways to intelligently override currently active contexts, policies and actions in order to increase interoperability in a pervasive system. This approach is orthogonal to ours, as we aim at providing a flexible policy framework which can be adapted to the needs different applications. An automatic adaptation of individual policies at run time is not in our focus, as we deem the thread of increased complexity and policies which cannot be validated as too high.

3 The Apollon Policy Framework

In this section, we introduce the basic concepts of the Apollon policy framework. We will start by giving an overview of the main components of the framework in subsection 3.1 and continue with a description of the policy decision process as it is currently realized in the framework. After that, we will explain the structure of policy modules, which can be loaded into the framework in order to add support for further high-level policies.

3.1 Framework Components

Figure 1 provides an overview of the main components of the Apollon framework. The core components are a semantic knowledge base in which the policy model is represented and a decision engine which evaluates access requests and events against the policies stored in that knowledge base. The knowledge base is realized in an individual component which manages a set of OWL2 ontologies and provides reasoning functions to the other components of the framework. The decision engine maintains the actual policies and evaluates access requests by following the decision process described in 3.2. Any input to the decision engine comes from event listeners and policy enforcement points (PEP). Event listeners can react on any kind of event, such as devices joining and leaving the network or any contextual changes, for example. PEPs intercept outgoing and incoming service calls and responses (i.e., a single message round-trip involves four PEPs), convert them in an access request format and then send them to the decision engine and enforce the resulting decision. They are supported by annotation handlers, which add metadata to an access request, containing information which is needed by the decision engine. Which types of event listeners will be useful and how PEPs will be integrated into an existing service architecture depends on the underlying middleware and is not predefined by Apollon. However, for illustrating a possible realization of these components, we describe an implementation based on the OSGi in section 6. Policy modules can be added to the decision engine in plug-in manner. Their purpose is to enrich the expressiveness of policies by extending the existing policy model or by adding evaluation functions to the decision engine. As policy modules can depend on each other, it is possible to build increasingly abstract models based on previously loaded modules. The benefit of this modular approach is thus twofold: by removing unneeded models, the footprint of the Apollon framework can be scaled to the actual needs of an application and by adding modules building on other existing modules, more abstract and easy to understand policies can be realized. The details of policy modules are covered in subsection 3.3 and in section 4 we show how an increasingly abstract policy model can be built.

3.2 Decision Engine

The decision engine’s task is to receive access requests via PEPs and events via Event Listeners, evaluate the policy against the incoming request and either return the policy decision to the PEP or invoke an obligation. For the sake of extensibility, the decision process has been structured by four different phases, through which the engine passes for every policy decision: Configuration, Decision, PostDecision, and Classification. Policy modules can contribute to each of these phases and thereby extend the expressivity of the policy language. The first phase, Configuration, is only passed when initially loading the policy file and set up configurations as required by the policy modules. For synchronously (i.e., while the PEP blocks) evaluated access requests the Decision phase follows. In this phase, the engine decides whether the access should be granted or not by evaluating the access request, its metadata, the policy model stored in the knowledge base and specific rules. As a result of this phase, a single decision, being either permit, deny or undefined is found and passed on to the next
phase, called PostDecision. In that phase, policy modules cannot modify the decision anymore, but merely log access requests and resulting decisions – for example in order to realize history-based access control models. As a final step in the synchronous evaluation branch, the Classification phase is passed. During this phase, policy modules can annotate the decision which has been taken. As an example, this phase can be used to annotate access control decisions with metadata for declaring policy declaring composition strategies, as used in (?). Besides access requests which are evaluated in a synchronous way, Apollon also supports asynchronously evaluated event-action policies. They are triggered by events which are received by event listeners and routed into the decision engine. In that case, the decision engine passes the Event phase where the event is evaluated and matching rules and conditions from the policy are evaluated. If the policy specifies any obligations which must be enforced upon the occurrence of the received event, the respective actions are executed during the following Action phase. After the decision engine has passed these phases, it returns to the state in which it was after the Configuration phase and is ready to accept further access requests and events. This decision process is part of the Apollon core and is not supposed to be modified. Policy modules can make contributions to each of these phases in the form of plug-ins, as described in the next subsection.

3.3 Policy Modules

Thus, Apollon allows to extend the basic authorization model from section 4 by further high-level policy models which are easier to understand for developers. In particular, policy modules can make use of the features provided by other modules and thus support increasingly abstract models, as illustrated below. A policy module comprises a set of components, called plug-ins, which can add facts to the knowledge base or extend the decision process by contributing to the aforementioned phases. The most important plug-in types supported by the framework are as follows: Ontology Fragments extend the knowledge base used by the decision engine and thereby increase the expressivity of policies. Due to the monotonicity property of description logics, policy modules cannot retract or overwrite parts of the already existing knowledge base but merely add additional facts. That way, it is not possible to accidentally overwrite facts in the ontology which have been defined by another module. Policy Fragments specify DTDs of XML fragments which can be used in the policy specification. It is not always feasible or sensible to express everything in the ontology. Especially information which is required for the evaluation of an access request but is not related to the facts in the knowledge base should be kept in separate XML files, as it would otherwise only increase the size and complexity of the ontology, and along with that, increase the reasoning overhead exponentially. As an example, we use policy fragments to specify configuration parameters of the core policy module, such as the mapping from attributes in an access request to the values of subject, resource and action, as they are evaluated during the policy decision. Decision Plug-ins are called during the Decision phase, evaluate access requests, usually with involvement of the knowledge base and return a deny or permit decision. If multiple decision plug-ins return contradicting decisions, the final decision is selected by a combining algorithm, as shown in section 4. Although the exemplary policy models below are almost fully based on description logics, i.e. represented in the ontology, not all evaluation algorithms can be written purely in DL and decision plug-ins might therefore add pieces of custom code which goes beyond the expressivity of DL in order to evaluate an access request. PostDecision Plug-ins are called during the PostDecision phase and get passed the decision along with the
original access request. As stated above, they cannot modify the decision anymore, but may for example log the decision in order to realize a history-based access control model.

*Event Listeners* register themselves for event topics and subsequently receive events of these topics. Upon arrival of an event, they can perform any action that has been implemented, for example switch role memberships as in the DRBAC (*dynamic RBAC*) model in section 5 or to trigger the execution of obligations. Thus, a policy decision may not be necessarily initiated by an access request, but can also be the result of an asynchronous event.

*Classification Plug-ins* annotate policy decisions with meta data, which can be used in order to distinguish different types of decisions. For example, it would be possible to classify decisions into different strengths, depending on whether they have to be enforced under all circumstances or are allowed to be overridden by other PDPs.

### 4 A Basic Policy Model

In the previous chapter we introduced the software components and the policy decision process of Apollon. In this chapter, we will now describe an exemplary basic policy model supported by Apollon and show how further, more high-level models can be realized on top of it. Description logics was chosen for formalization, as it is the underlying logic of OWL, which is commonly used to model domain knowledge in pervasive systems. Thus, by implementing the policy model in OWL, already existing domain knowledge can easily be integrated into security policies. Further, the implementation in OWL is kept close to description logic formalization and thereby facilitates checking the implementation for correctness. A further benefit is that standard semantic web reasoners can be used for evaluating and analyzing policies. As one of the main challenges in the context of security policies is to make policies understandable for non-security experts, we expect a DL-based policy model, and the analysis and explanation features that come with it, to receive better user acceptance.

However, as stated in (Toninelli et al., 2005), DL alone is not expressive enough, for example because it lacks support for variables, does not support non-monotonic reasoning, and, as a result, does not allow negation by failure. Therefore, we will complement the DL model by the decision plug-ins from section 3.3 in order to implement a hybrid policy decision process. The basic authorization module provides basic access control rules based on subjects, resources and actions, similar to that of XACML. These rules are formalized as follows (for details on the notation we refer to (Baader et al., 2003) and (Baader et al., 2007)):

\[
\begin{align*}
\text{Policy} & \equiv \forall \text{hasRule} \cdot \text{Rule} \sqcap \text{hasRulePref} \cdot \text{String} \\
\text{Rule} & \equiv \forall \text{hasSubject} \cdot \text{Subject} \sqcap \forall \text{hasResource} \cdot \text{Resource} \\
& \quad \forall \text{hasAction} \cdot \text{Action} \\
& \quad \forall \text{hasEffect} \cdot \text{Effect} \\
& \quad = 1 \text{hasNumber} \\
\text{Effect} & \equiv \{\text{deny}, \text{permit}\} \sqcap \text{Obligation}
\end{align*}
\]

A policy comprises a set of rules and a rule preference, determining which rule should be preferred in the case of contradicting rules. The rules assign either *deny* or *permit* and an optional obligation to a triple of Subject, Resource and Action and are ordered by assigning distinct numbers to them, using the `hasNumber` relation (distinction of the number cannot be expressed in DL and has to be checked by the decision plug-in). An access request as received by the decision engine consists of a description of the subject which initiated the request `s ∈ Subject`, the resource which is to be accessed `r ∈ Resource`, and the action which is to be performed on the resource `a ∈ Action`. Given the request and a set of rules `Rule`, the decision engine returns an effect `e ∈ Effect`, depending on the set of applicable rules and the rule preference, as follows: a rule `rule` is applicable if

\[
\begin{align*}
\text{rule} & \in \{\text{Rule} \sqcap \text{hasSubject} \cdot \{s\} \sqcap \\
& \quad \text{hasResource} \cdot \{r\} \sqcap \\
& \quad \text{hasAction} \cdot \{a\}\}
\end{align*}
\]

and the final decision is selected among them according to the rule preference, being either `first`, `last`, `permit` or `deny`. When using the `first preference`, the decision engine selects the effect of the first rule out of the set of all applicable rules, i.e. `e₁` is chosen as effect if `r₁ ∈ \{\text{Rule} \sqcap \text{hasEffect} \cdot \{e₁\} \sqcap \text{hasNumber} \cdot \{x\}\}` and `x` less than the number of all other applicable rules. The `last` preference is applied likewise, selecting the last applicable rule, respectively. The `permit` preference selects the first applicable rule `r` with `r ∈ \{\text{Rule} \sqcap \text{hasEffect} \cdot \{\text{permit}\}\}` otherwise returns `deny` and the `deny` preference acts equally, selecting the first denying rule.

A detailed discussion of obligations is omitted here, as it is not in main focus of the paper. In general, an obligation will specify an action that has to be carried out whenever the respective rule has been selected by the decision engine. The obligation is carried out by the PEP before the actual decision is applied, otherwise the access request has to be refused,
regardless of the policy decision. In the scope of the OSGi-based prototype, as described in section 6, such an obligation refers to the URL of an OSGi bundle containing the respective action.

5 Dynamic RBAC

Based on the basic authorization rules, more complex access control modules can be added. As an example, we show how the Apollon framework can be used to realize a dynamic role-based access control model (DRBAC). Role-based access control uses roles to decouple subjects and permissions, which results in a model that is easy to maintain and in most cases maps better to organizational security policies. Further, it is possible to specify separation-of-duty constraints in order to guarantee that a subject cannot be assigned to conflicting roles coevally. Many frameworks for RBAC exist and a plethora of slightly different interpretations of the model are in use (e.g., (Bacon et al., 2002; OASIS, 2005a; Alm et al., 2009)). This purpose of this section is thus not to add yet another RBAC implementation but to illustrate how increasingly abstract policy models can be realized in Apollon.

In this example, we will realize a simple DRBAC model with hierarchical roles. The respective policy module will comprise an ontology fragment, modeling the policy structure, an event adapter, which triggers the activation of rules, a retrieval plug-in which allows using roles as attributes of a subject and a decision plug-in for evaluation of RBAC policies.

5.1 Mapping DRBAC to OWL

Various authors have proposed a representation of RBAC in OWL and while most of the suggested approaches are feasible, they have different drawbacks which we aim to overcome. In (Finin et al., 2008), two approaches are proposed and compared: the first one models subjects as individuals and roles as classes. Role membership is then modeled by assigning a subject to the classes of its active roles. The second approach models roles as individuals and assigns them to subjects using a hasRole property, but has the drawback that additional rules are needed to express role hierarchies. The authors of (Finin et al., 2008) consider the first approach more attractive as it provides more options to analyze the model (such as identifying all subjects with certain permissions) and allows to directly evaluate role hierarchies by using subsumption checking. However, as identified in (Ferrini and Bertino, 2009), the deficit of the first approach is that it does not allow to define separation-of-duty constraints over hierarchical roles as this would result in an inconsistent model (two hierarchical classes would be modeled as being disjunct, which is not feasible). In (Ferrini and Bertino, 2009), the authors try to overcome the deficits from (Finin et al., 2008) by adapting the second approach such that separation-of-duty can be checked using a standard reasoner. However, the model proposed in (Ferrini and Bertino, 2009) assigns permissions directly to roles instead of, which does not comply with the core RBAC specification from (Ferraiolo et al., 2001) we wanted to implement in this case. Nevertheless, in essence we adopt the representation from (Ferrini and Bertino, 2009) and adapt it to meet the RBAC specification as follows.

We extend subjects in the basic model from section 4 by a User class. Users are defined as individuals of that class and assigned to roles by a hasActiveRole property. By changing this assignment from hasActiveRole to hasRole, the respective role can be deactivated for the user. Role hierarchy is implemented, as proposed in (Ferrini and Bertino, 2009), by subRoleOf and superRoleOf properties. In contrast to (Ferrini and Bertino, 2009), we do not require dedicated rules to evaluate hierarchical roles but rather define a property chain hasActiveRole \circ subRoleOf \rightarrow hasActiveRole. This way, the reasoner can infer role membership of a subject for all its super roles, as depicted by Figure 3. Permissions, finally, consist of an action and a resource and are assigned to roles by a hasPermission property. To summarize, the ontology fragment of the DRBAC module is defined as in the following equations. Note that due to usage of property chains, access requests can be decided in the DRBAC model using only description logic – the only part which has to be done outside the ontology is the activation of roles.

\[
\begin{align*}
\text{User} & \sqsubseteq \text{Subject} \\
\text{User} & \equiv \forall \text{hasRole}.\text{Role} \\
\text{Role} & \equiv \forall \text{hasPermission}.\text{Permission} \\
& \quad \forall \text{subRoleOf}.\text{Role} \\
& \quad \forall \text{hasActiveRole}.\text{ActiveRole} \\
\text{Permission} & \equiv \forall \text{hasResource}.\text{Resource} \\
& \quad \forall \text{hasAction}.\text{Action} \\
\text{ActiveRole} & \sqsubseteq \text{Role} \\
\text{superRoleOf} & = (\text{subRoleOf})^- \\
\text{hasActiveRole} & \leftarrow \text{hasActiveRole} \circ \text{subRoleOf} \\
\text{subRoleOf} & \sqsubseteq \text{subRoleOf} \circ \text{subRoleOf}
\end{align*}
\]
5.2 Implementing Separation of Duty

A common use case of RBAC models is to apply separation-of-duty constraints in order to avoid that users adopt conflicting roles, such as the roles of an applicant and a funding body. Depending on whether these roles must only not be adopted at the same time or must never be assigned to the same subject at all, the terms dynamic (DSoD) or static separation of duty (SSoD) are used. As in (Ferrini and Bertino, 2009), the authors have proposed an OWL implementation of both types, which we can adopt without changes, we refer to (Ferrini and Bertino, 2009) for more details and provide only the formalization of the applicant/funder example as a DSoD:

\[
\begin{align*}
DSoD_1 & \equiv \text{hasActiveRole}(\text{applicant}) \\
DSoD_2 & \equiv \text{hasActiveRole}(\text{funder}) \\
\emptyset & \equiv DSoD_1 \land DSoD_2
\end{align*}
\]

5.3 Using the DRBAC model in Apollon

To finally make use of the DRBAC model in the Apollon framework, two components are required: an event adapter for activating roles and a decision plug-in to actually evaluate access requests.

The event adapter simply accepts events containing a subject, a role and a Boolean value for activation/deactivation. Whenever such an event \( (\text{subject}, \text{role}, \text{activate}) \) is received, the event adapter checks whether \( \text{role} \) is actually a role of \( \text{subject} \) by verifying if there is a property \( \text{hasRole}(\text{subject}, \text{role}) \) in the ontology and then activates the role by adding a \( \text{hasActiveRole}(\text{subject}, \text{role}) \) axiom (or removes it in case of deactivation). At that point, it has to verify that the role activation does not violate DSoD constraints. This can simply be done by checking the ontology for consistency, because in case multiple incompatible roles would be activated for a subject, it would become an instance of both disjoint classes \( DSoD_1 \) and \( DSoD_2 \) at the same time, which would result in an incompatible ontology. In that case, the event adapter will refuse to activate the role.

The event adapter is generic and independent from any specific application, so we do not make any assumptions about how these events are triggered at this point. Apollon also supports event-condition-action (ECA) policies, which can be used to trigger role activations. As ECA policies are not in the scope of this paper, we omit details and simply assumed that such events are emitted from external components, as in the exemplary implementation described in section 6.1.

As the DRBAC policies are fully represented in OWL, the functionality of the decision plug-in is limited to using the reasoner for the following queries: whenever an access request is to be evaluated, subject, action and resource are extracted from its attributes and the ontology is queried for subjects with activated roles which match the requested permission, by inferring instances of the following class expression:

\[
\begin{align*}
\{\text{subject}\} \text{ that hasActiveRole} \\
\text{some (ActiveRole that hasPermission} \\
\text{some (Permission} \\
\text{that hasAction value action} \\
\text{and hasResource value resource)}
\end{align*}
\]

If this DL query returns any results, the DRBAC model grants the access request, otherwise it refuses it. The final decision of the request however depends not only on the DRBAC module but on all other involved policy modules. Only if all of them grant the access (as long as they are not indefinite about the request), the Apollon framework actually grants access to the requested resource.

6 Implementation Issues

A prototype of the Apollon framework has been implemented in order to test the concepts described in this paper and to serve as a basis for our future work. As an underlying middleware we used OSGi and connected peers to each other using R-OSGi\(^1\). This platform was chosen as we consider it a good representative of a typical pervasive system and because it provides a modularization layer which facilitates the modular implementation of Apollon. OSGi has many applications in resource-constrained settings, for example in vehicle applications\(^2\) and R-OSGi is a lightweight distribution mechanisms for OSGi bundles which has already been used in pervasive systems like flowSGI\(^3\), Hydra\(^4\), etc.

---

\(^{1}\)http://r-osgi.sourceforge.net/

\(^{2}\)http://www.osgi.org/VEGi/

\(^{3}\)http://flowsigi.inf.ethz.ch/

\(^{4}\)http://www.hydramiddleware.eu
6.1 PEPs, Annotations and Event Listeners

An application developer who wishes to use Apollon needs to implement two platform-specific components which need to be integrated into the underlying middleware and are thus only provided as empty stubs by Apollon – PEPs and Annotation Handlers. PEPs must be attached to outgoing and incoming service calls and Annotation Handlers must collect information necessary to take a policy decision and provide it to the PEPs.

As for PEPs, we used OSGi hooks for attaching them to remote services in our platform. OSGi hooks are a mechanism that allows us to intervene whenever a service registers itself in the OSGi platform and announces that it should be published over R-OSGi. If that happens, the service is replaced by a proxy service, which includes calls to the PEPs for every incoming and outgoing request and does otherwise merely forward the call to the original service. This way, it is ensured that PEPs are involved in every service invocation. The benefit of this approach is that it is completely transparent for all other services, so that it allows to “policy-enable” already existing applications based on R-OSGi without the need to modify any existing implementation.

The annotation handlers in our prototype work at Java level only, that is, they collect information about the calling class, method and thread and annotate each service call with it. In a productive set-up, additional annotation handlers, especially for collecting context information such as the current location of a (mobile) service, and environmental factors like temperature, time and lighting would provide much more valuable information for policy decisions. However, whether such information makes sense and how exactly it should be collected and represented is application-specific and was thus not in the scope of the prototype.

As for event listeners, as used for role activation in the DRBAC model, we rely on the event management of OSGi and for testing “situation-specific” policies, relying on high-level events that indicate begin and end of a certain situation, we used the Esper complex event processing (CEP) engine. Complex events as detected by the CEP engine were then used to trigger role changes in the DRBAC model. At this point, the event-condition-action (ECA) model of Apollon will be integrated as part of our future work, so as to allow defining and reasoning over ECA policies.

6.2 Decision Engine and Reasoning

Unlike the components described in the previous subsection, components such as the decision engine, the knowledge base and the policy modules are independent from the underlying middleware. The knowledge base has been realized as a set of OWL 2 files which are provided by the individual policy modules. A BasicSemanticModule provides the basic authorization policies and in addition a retrieval plug-in that allows to use class expressions in Manchester DL syntax (Horridge et al., 2006) within the specification of attributes. Listing 1 shows how such these class expressions can be used in a policy. Here, a class expression is used to refer to all subjects which are calling from a certain Java method. The usefulness of this specific condition left aside, it shows how the usage of semantic information facilitates the specification of more abstract and understandable conditions.

Listing 1: Basic authorisation rule using class expressions

```xml
<authorisationRule>
  <AccessType>req.incoming</AccessType>
  <Subject>
    <sem:expression>
      Subject THAT hasJavaMethod VALUE "org.apollon.test.Client.call"
    </sem:expression>
  </Subject>
</authorisationRule>
```

In addition to the BasicSemanticModule, we have implemented an RBACPolicyModule which provides the RBAC model, and a DRBACModule which extends the functionality of the previous two modules by the capability of changing roles upon events. As a reasoning engine we used Pellet 2.06 along with OWLAPI7 for handling the ontologies. The individual decision modules are realized as OSGi services so they can be loaded and unloaded at run time, without restarting the whole framework. Using the dependency management of OSGi we can ensure that policy modules are only registered in the framework when all their dependencies are fulfilled, i.e. as soon as the BasicSemanticModule is unloaded, RBAC and DRBAC become unavailable as well.

6.3 Run time behavior

As the complexity of reasoning over OWL 2, which relates to the $\mathcal{SHOIN}(D)$ logic, is up to NexpTime,

5http://esper.codehaus.org/

6Newest Pellet version is 2.2.2, cf. http://clarkparsia.com/pellet

7http://owlapi.sourceforge.net/
it is interesting to verify whether an ontology-based policy decision engine would achieve satisfying performance for practical use. Although an in-depth performance analysis of Apollon is out of the scope of this paper, we will provide some run time results of the prototype. For this purpose, we measured the computing time for deciding access requests using the DRBAC model introduced above. The experiments were made on an Intel Core 2 Duo 2GHz machine, running Ubuntu 10.04 and Sun Java 1.6.0.22, leaving Pellet’s default optimization settings untouched. The first ontology we tested (“small”) is comparatively sparsely populated with 6 roles, 4 users and 3 permissions. After classification, this results in an ontology with 88 individual axioms, 13 class axioms, 28 object properties and 7 data properties. The second ontology (“large”) reflects a policy of more realistic size with 205 roles, 1003 users and 200 permissions. It results in a classified ontology with 4279 individual axioms, 19 class axioms, 46 object properties and 7 data properties. Both ontologies are of ALCO1(D) expressivity.

Our experiments have shown that for the small ontology, an average response time of 5.04 ms was achieved, where the first request takes up to 6.96 ms while further requests decrease down to 1.02 ms. As for the large ontology, the average response time was 20.47 ms, where the first request amounts to 25.02 ms and further requests can get as fast as 10.87 ms. These values show that even for policies of reasonable size, as they might be used in practice, the time required for deciding access requests is satisfying and we can consider Apollon with the exemplary DL-based model to be suited for use in real-world applications.

7 Conclusion and Future Work

In this paper we have introduced the Apollon framework, a policy framework for pervasive systems which makes extensive use of ontologies for representing and reasoning over security policies. Apollon has been built to meet in particular the challenges of pervasive systems, stated in the introduction of this paper: by describing entities in DL and in combination with the easy-to-write Manchester DL syntax, policy specification is facilitated and the actual intent (in terms of the implemented security model) becomes more visible. The OWL representation of the exemplary policy model described in this paper allows us to separate the actual policy (reflecting the security model) from domain knowledge (reflecting assumptions about security mechanisms and devices). Further, by example of a DRBAC model, we have shown that DL allows to reason about policies and to verify security properties like SoD. However, we acknowledge that DL alone is not expressive enough for most policies and should thus mainly be used for modeling domain knowledge and reasoning over policies rather than performing the actual policy decision process.

Due to the modular software architecture we propose, it is possible to only load the required policy modules, thereby reducing the footprint and complexity of the policy framework to the actually needed functionality. We have shown how basic attribute-based access control rules can be realized and how a dynamic RBAC model can be implemented on top of them by adding a policy module. As part of our future work, we will develop further access control models for Apollon, which take into account context-specific access rights, add features for security negotiations between peers in order to support self-protecting systems and continue our research on policy analysis based on OWL and reasoning. Furthermore, the implementation of a development GUI for easy policy administration is ongoing.

REFERENCES

Boyle, J., Cohen, R., Herzog, S., Rajan, R., and Sas-


OASIS (2005a). Core and hierarchical role based access control (rbac) profile of xacml v2.0. OASIS.


