APOLLON: TOWARDS A SEMANTICALLY EXTENSIBLE POLICY FRAMEWORK

Julian Schütte
Fraunhofer Institute for Secure Information Technology SIT, Parkring 4, Garching, Munich, Germany

Keywords: Access control, Semantic web, Software architecture.

Abstract: Pervasive systems with ad hoc connectivity and semantic service discovery are a challenging environment when it comes to dynamically managing access rights and security settings. 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 predetermined 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. In this paper, we present the software architecture of Apollon, and show by the example of a DRBAC-model how the expressiveness of Apollon can be successively 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 at run time. While more and more applications based on such systems occur, controlling access rights and security settings throughout the whole application is still a challenge. Some of the open issues in that area are:

Most frameworks (Twidle et al., 2009; Lalana Kagal, 2006; Uszok et al., 2003) 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. Furthermore, there is a demand to express high-level security models. Many policy languages feature low-level and fine-granular rules, which often results in the actual security model being buried in a huge set of complex rules, mixing up the 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. In section 2, we introduce the Apollon framework, the policy decision engine, as well as the concept of using DL to represent security policies. An exemplary basic authorization model, based on DL is then introduced in section 3 and an example of a high-level security model on top is given in section 4, along with results from our prototype implementation. Section 5 concludes the paper and outlines future work.

Schütte J..
APOLLON: TOWARDS A SEMANTICALLY EXTENSIBLE POLICY FRAMEWORK.
DOI: 10.5220/0003523603910395
In *Proceedings of the International Conference on Security and Cryptography* (SECRYPT-2011), pages 391-395
ISBN: 978-989-8425-71-3

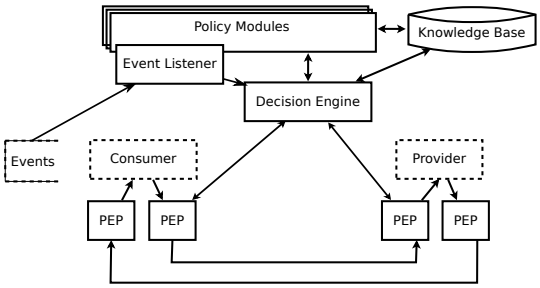

Figure 1: High-level overview of framework components.

2 THE APOLLON POLICY FRAMEWORK

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 and events by following the decision process described below. Any decision is triggered by either *event listeners* or *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. How event listeners and PEPs have to be integrated into an existing middleware is not specified by Apollon, however, in our prototype we use the OSGi *event admin* service as an event listener and *service hooks* to non-intrusively attach PEPs to any OSGi service.

Policy Modules. The decision engine can be extended in a plug-in manner by means of *Policy modules*. A policy module can contribute to any of the decision phases, described in the following paragraph, by extending the policy specification or decision logic itself, by adding facts to the knowledge base, registering further event listeners or by providing a composition strategy for merging policy decisions from multiple domains. Due to the monotonicity property of DL, facts can be added to the knowledge base without invalidating any of the conclusions taken before. Policy modules can depend on each other and thus serve as building blocks for increasingly abstract policy 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.

Decision Process. The decision engine is responsible for reacting to incoming events or access requests from PEPs, evaluate policies and return a decision or trigger the execution of obligations. For the sake of extensibility, the decision process has been structured by different phases, which the engine passes for every policy decision and which we will only sketch here, due to lack of space. For synchronously evaluated access requests the engine decides in a *Decision* phase 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, containing either *permit* or *deny* as an effect and a set of high-level obligations 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 with metadata, for example to implement composition strategies, as used in (Lee et al., 2006). Besides access requests which are evaluated in a synchronous way, Apollon also supports asynchronously evaluated ECA policies which are the basis for situation-specific access rights and obligations. They are triggered by events which are received by event listeners and routed into the decision engine, which then evaluates the policy's ECA rules. If the policy specifies any obligations which must be enforced upon the occurrence of the received event, the respective high-level actions are refined to applicable mechanisms and executed during the following *Action* phase. 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 A BASIC POLICY MODEL

We will now describe an exemplary basic policy mo-

del 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 2 in order to implement a hybrid policy decision process. The basic authorization module provides simple 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., 2007)):

$$\begin{aligned} Policy &\equiv \forall hasRule.Rule \sqcap hasRulePref.String \\ Rule &\equiv \forall hasSubject.Subject \sqcap \\ &\quad \forall hasResource.Resource \sqcap \\ &\quad \forall hasAction.Action \sqcap \\ &\quad \forall hasEffect.Effect \sqcap \\ &\quad = 1\, hasNumber.n \\ Effect &\equiv \{deny, permit\} \sqcap Obligation \end{aligned}$$

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, Action and are ordered by assigning distinct numbers to them, using the hasNumber relation. An access request consists of a description of the subject which initiated the request $s \in (Subject)^I$, the resource which is to be accessed $r \in (Resource)^I$ and the action which is to be performed on the resource $a \in (Action)^I$. Given the request and a set of rules $(Rule)^I$, the decision engine returns an effect $e \in (Effect)^I$, depending on the set of applicable rules and the rule preference, as follows: a rule *rule* is applicable if

$$\begin{aligned} rule \in\ & (Rule \sqcap hasSubject.\{s\} \sqcap \\ & hasResource.\{r\} \sqcap \\ & hasAction.\{a\})^I \end{aligned}$$

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. e_f is chosen as effect if $r_1 \in (Rule \sqcap hasEffect.\{e_f\} \sqcap hasNumber.\{x\})^I$ 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 $r \in (Rule \sqcap hasEffect\{permit\})^I$, 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 the main focus of the paper. In general, obligations specify actions which have to be carried out whenever a respective rule has either been triggered by an access request or asynchronously by an event. In case of access requests, 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 current OSGi-based prototype, such an obligation refers to the URL of an OSGi bundle that implements the respective action.

4 DYNAMIC RBAC

To show how more complex access control models can be built by means of additional policy modules, we describe a dynamic role-based access control model (DRBAC) with hierarchical roles that takes into account separation-of-duty (SoD) 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 (Bacon et al., 2002; OASIS, 2005; Becker and Sewell, 2004). The 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.

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.

Mapping DRBAC to OWL. Various authors have proposed representations 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 and allows to directly evaluate role hierarchies 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 SoD 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 SoD can be directly checked by a standard reasoner. However, the proposed model assigns permissions directly to subjects instead of roles, which does not comply with the core RBAC specification from (Ferraiolo et al., 2001) we wanted to implement in this case. We therefore adopt in essence the representation from (Ferrini and Bertino, 2009) and adapt it to meet the RBAC specification as follows. We extend *subjects* from the model in section 3 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 memberships of a subject for all its super roles, as depicted by Figure 2. Permissions, finally, consist of an action and a resource and are assigned to roles by a hasPermission property. To summarize, the ontology 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 DL, and only the activation of roles has to be done from outside the ontology.

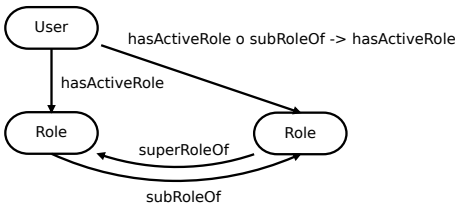

Figure 2: Property chains for role hierarchies.

$$
\begin{aligned}
User &\sqsubseteq Subject \\
User &\equiv \forall hasRole.Role \sqcap \\
Role &\equiv \forall hasPermission.Permission \sqcap \\
&\quad \forall subRoleOf.Role \\
&\quad \forall hasActiveRole.ActiveRole \\
Permission &\equiv \forall hasResource.Resource \sqcap \\
&\quad \forall hasAction.Action \\
ActiveRole &\sqsubseteq Role \\
superRoleOf &= (subRoleOf)^{-} \\
hasActiveRole &\leftarrow hasActiveRole \circ subRoleOf \\
subRoleOf &\sqsupseteq subRoleOf \circ subRoleOf
\end{aligned}
$$

Adding 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:

$$
\begin{aligned}
Dsod_1 &\equiv hasActiveRole.\{\texttt{applicant}\} \\
Dsod_2 &\equiv hasActiveRole.\{\texttt{funder}\} \\
\emptyset &\equiv Dsod_1 \sqcap Dsod_2
\end{aligned}
$$

Runtime Results. As the complexity of reasoning over OWL2 is up to NexpTime, it is interesting to test whether an ontology-based policy decision engine achieves satisfying performance for practical use. We measured the computing time for deciding access requests using the DRBAC model with a reasonably populated ontology with 205 roles, 1003 users and 200 permissions, which has $\mathcal{ALCOI}(D)$ expressivity and results after classification with the Pellet reasoner in 4279 individuals, 19 classes, 46 object properties and 7 data properties. The average response time for

an access request was 20.47 ms, where the first request amounts to 25.02 ms and further requests can get as fast as 10.87 ms[1]. This shows that even for policies of reasonable size, the time required for deciding access requests is satisfying and the DL-based DRBAC model can be considered to be suited for use in real-world applications.

5 CONCLUSIONS AND FUTURE WORK

We have introduced the Apollon framework, an extensible policy framework which makes 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 easy-to-write syntaxes such as Manchester DL or Turtle, policy specification is facilitated and the author's actual intent, in terms of the security model, becomes more visible. The DL 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 reasoning can be used to decide acces requests and 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.

The modular software architecture allows to only load the required policy modules, thereby reducing the footprint of the policy framework to the actually needed functionality. As part of our future work, we will 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.

REFERENCES

Baader, F., Horrocks, I., and Sattle, U. (2007). *Handbook of Knowledge Representation*, chapter 3 Description Logics, pages 135–180. Elsevier. ISBN 0444522115.

Bacon, J., Moody, K., and Yao, W. (2002). A model of oasis role-based access control and its support for active security. *ACM Trans. Inf. Syst. Secur.*, 5:492–540.

Becker, M. Y. and Sewell, P. (2004). Cassandra: Distributed access control policies with tunable expressiveness. In *Proc. 5th IEEE Int'l Workshop on Policies for Distributed Systems and Networks (POLICY)*, pages 159–168. IEEE Computer Society.

Ferraiolo, D. F., Sandhu, R., Gavrila, S., Kuhn, D. R., and Chandramouli, R. (2001). Proposed NIST Standard for Role-Based Access Control.

Ferrini, R. and Bertino, E. (2009). Supporting rbac with xacml+owl. In *Proceedings of the 14th ACM symposium on Access control models and technologies (SACMAT '09)*, pages 145–154, New York, NY, USA. ACM.

Finin, T., Joshi, A., Kagal, L., Niu, J., Sandhu, R., Winsborough, W. H., and Thuraisingham, B. (2008). ROWLBAC - Representing Role Based Access Control in OWL. In *Proceedings of the 13th Symposium on Access control Models and Technologies*. ACM Press.

Lalana Kagal (2006). The Rein Policy Framework for the Semantic Web. http://dig.csail.mit.edu/2006/06/rein/.

Lee, A., Boyer, J. P., Olson, L. E., and Gunter, C. A. (2006). Defeasible security policy composition for web services. In *Proceedings of the fourth ACM workshop on Formal methods in security*, FMSE '06, pages 45–54, New York, NY, USA. ACM.

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

Toninelli, A., Bradshaw, J. M., Kagal, L., and Montanari, R. (2005). Rule-based and ontology-based policies: Toward a hybrid approach to control agents in pervasive environments. In *Proc. of the Semantic Web and Policy Workshop*.

Twidle, K., Dulay, N., Lupu, E., and Sloman, M. (2009). Ponder2: A policy system for autonomous pervasive environments. In *The Fifth International Conference on Autonomic and Autonomous Systems (ICAS)*, pages 330–335. IEEE Computer Society Press.

Uszok, A., Bradshaw, J. M., Jeffers, R., Suri, N., Hayes, P. J., Breedy, M. R., Bunch, L., Johnson, M., Kulkarni, S., and Lott, J. (2003). Kaos policy and domain services: Toward a description-logic approach to policy representation, deconfliction, and enforcement. In *Third International Workshop on Policies for Distributed Systems and Networks (POLICY)*, pages 93–96.

[1]On Intel Core 2 Duo 2GHz, Ubuntu 10.04, Sun Java 1.6.0.22, leaving Pellet's default optimization settings untouched.