Toward Authorization as a Service

A study of the XACML standard

Romain Laborde, François Barrère, Abdelmalek Benzekri
IRIT/SIERA
University Paul Sabatier
Toulouse, France
{laborde,barrere,benzekri}@irit.fr

Keywords: Authorization as a Service, XACML, Self-contained policy, OSGi

Abstract
Cloud computing has promoted the notion of service as the leading way to deliver and consume computing resources. Today, security is going down that road and the term security as a service is emerging. Authorization that consists in managing permissions is one of the main classic security services. We propose in this article to study how authorization could be delivered/consumed as a Service. We focus on the XACML standard that has been adopted by the cloud security community because of its native flexibility and adaptability properties. Although XACML seems to fulfill the requirements of authorization as a Service in theory, it is very complex to realize it in practice. We propose a service oriented component architecture together with the concept self-contained policy to cope with this issue. This approach allows both the cloud consumers to adapt the authorization system to their authorization policies and the cloud providers to minimize the cost of providing a flexible authorization service.

1. INTRODUCTION

Cloud computing has promoted the notion of service as the leading way to deliver and consume computing resources. It is common to use applications supplied by a cloud provider, to store data out of our computer or to rent virtual machines for supporting our web site in our daily life now. Thereby, the number and nature of services provided are growing and become more diverse.

Today, security is going down that road. Security applications are migrating to the cloud. Existing examples are antiviruses that do no more run on local computers but on a cloud provider infrastructure. Identity service is another example. And more and more security applications are considered to turn out the same way. The Cloud Security Alliance summarizes it as Security as a Service (SecaaS) and provides the following definition: « Security as a Service refers to the provision of security applications and services via the cloud either to cloud-based infrastructure and software or from the cloud to the customers' on-premise systems » [1]. SecaaS looks to secure systems and data in the cloud as well as hybrid and traditional enterprise networks via cloud-based services.

The Cloud Security Association has also proposed in [2] a set of best cloud-related security practices divided into 14 domains. Domain 12, titled "Identity, Entilement and Access Management", recommend the use of XACML [3] as the authorization technology because it provides a declarative access control language and a policy-based management architecture. Using XACML is not mandatory for authorization management. However, it is clearly a choice of the Cloud Security Association to promote it. In Domain 14 that focuses on SecaaS, it is suggested that identity-as-a-Service may comprise an eco-system such as Policy Enforcement Point as a Service, Policy Decision Point as a Service and Policy Administration Point as a Service.

We propose in this article a service-oriented component software architecture of XACML that could be employed to implement Authorization as a Service. Especially, our goal is to design an XACML-compliant system that could be used as a service. One of the main challenges is to build a system that could easily be adapted/extended to service customers' requirements and that minimizes the cost of maintenance for the service providers at the same time. Our solution consists in a generic core authorization engine that can be extended by reusable components. We also exploit the concept of self-contained policy. This architecture has been implemented along the OSGi specification.

The rest of this article is organized as follows. Section 2 presents XACML and the issues related to extending and adapting XACML solutions. Section 3 introduces the concept of self-contained policy and our XACML service-oriented components architecture. Section 4 explains how to use this architecture to provide Authorization as a Service. Related works are listed in section 5. Finally, section 6 concludes the article and presents our future work.

2. OVERVIEW OF XACML 2.0

XACML (eXtensible Access Control Markup Language) is an XML-based language for access control that has been standardized in OASIS [3]. XACML describes both an attribute-Based access control policy language and a request/response language.

2.1. XACML Architecture

XACML provides a management architecture that describes the different entities and their roles related to the decision making process. A data-flow model describes this architecture. A simplified version of this model is depicted in Figure 2.

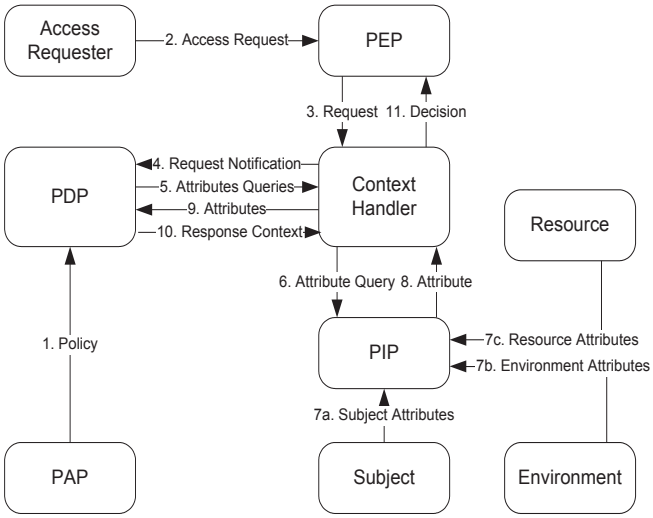

Figure 1. Simplified XACML data flow model

The model operates by the following steps.

1. Policy Administration Points (PAP) write policies and policy sets and make them available to the Policy Decision Point (PDP). These policies or policy sets represent the complete policy for a specified target.
2. The access requester sends a request for access to the Policy Enforcement Point (PEP).
3. The PEP sends the request for access to the context handler in its native request format, optionally including attributes of the subjects, resource, action and environment.
4. The context handler constructs a standard XACML request context and sends it to the PDP.
5. The PDP can request any additional subject, resource, action and environment attributes from the context handler if needed.
6. The context handler requests the attributes from a Policy Information Point (PIP).
7. The PIP obtains the requested attributes.
8. The PIP returns the requested attributes to the context handler.
9. The context handler sends the requested attributes. The PDP evaluates the policy.
10. The PDP returns the standard XACML response context (including the authorization decision) to the context handler.
11. The context handler translates the response context to the native response format of the PEP. The context handler returns the response to the PEP that enforces the authorization decision.

2.2. XACML policy language

XACML policy language is used to describe general access control requirements in term of constraints on attributes. Specifically, attributes could be any characteristics of the subject, resource, action, or environment in which the access request is made. Attributes have an identifier, which is a Uniform Resource Name (URN), and a data type also identified by a URN. Considering attributes makes the language very flexible. Moreover, XACML language is natively extensible. As a consequence, this standard is a good candidate for cloud-based security services implementations.

XACML defines standard attributes of subjects (e.g., subject-id, subject-category), resources (resource-id and target-namespace), actions (action-id, implied-action, action-namespace) and environment (current-time, current-date, and current-dateTime). However, non XACML standard attributes can be considered in an XACML policy by creating new URNs (see section 8 of [1]). Attributes are attached to standard XACML data types (such as Boolean, integer, string, etc) or any other user-defined data types can be created (such as "urn:ogc:def:dataType:geoxacml:1.0:geometry" defined by the Open Geospatial Consortium [4]).

Finally, an authorization policy rule consists in a Boolean expression on attributes and an associated decision (called effect). Attributes are manipulated using functions according to their data types. Like attributes and data types, the XACML specification has defined standard functions (e.g., Boolean and, or, string-equals, integer-greater-than, etc). But, it is possible to add new functions to extend XACML policies. For example, the Open Geospatial Consortium has defined a set of functions to manipulate geospatial attributes such as" "urn:ogc:def:function:geoxacml:1.0:geometry-contains").

2.3. XACML in practice

In order to understand the gap that exists between theory and practice, we present a simple example. In this example, we focus on the PDP issue only. We refer the reader to [5] for the PEPs issues.

Let consider a cloud provider that wants to develop Authorization as a Service for its customers. Respecting the CSA security guide, it provides an XACML authorization system. Now let consider a customer organization named MY-ORGANIZATION that wants a web service that provides maps. Maps include details on the organization's area. This organization wants to restrict the access of the organization's area details to its employees only. All the users' accounts are already stored in a MySQL database provided by the cloud provider. People who work for this organization can come from different companies (e.g., subcontractors). Thus, the users' account table contains a field, called *organization-name*, which value is the name of the users' origin.

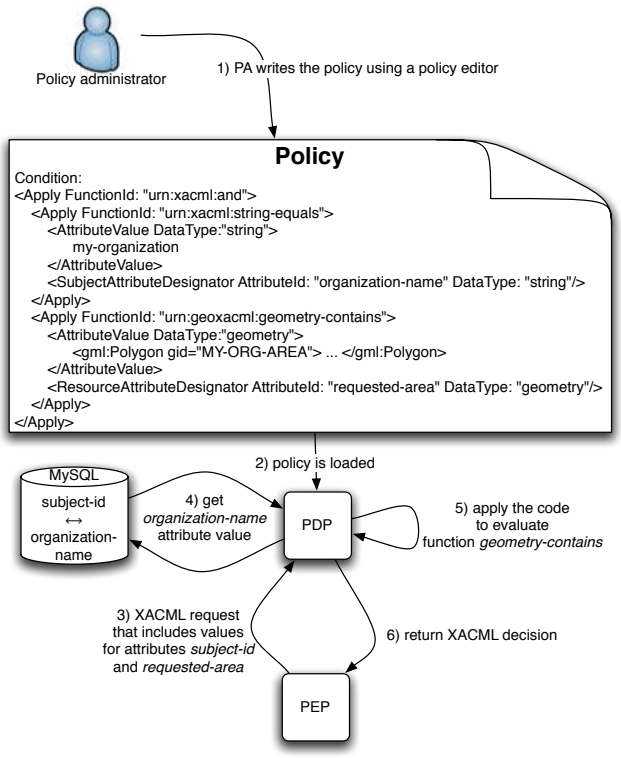

Figure 2. Example of XACML in practice

The MY-ORGANIZATION security administrator wants to make use of GEOXACML [4] that provides data types and operators to manipulate geo-data in XACML policies. The security policy is then: if *organization-name* of user is "my-organization" and *requested-area* is within MY-ORG-AREA then accept. Otherwise, reject the request. The security administrator writes this policy in XACML language (a sample is given in Figure 2). The execution process is the following (Figure 2):

1. The security administrator writes the policy in XACML
2. The policy is loaded in the PDP
3. When a user tries to access a map, the PEP catches the request and sends an XACML request to the PDP. This request includes the name of the user in subject attribute *subject-id* and the requested map in resource attribute *requested-area*. In this scenario, attribute *organization-name* is not provided by the PEP. We assume that the PEP sends attribute subject-id. The value of attribute organization-name, stored in a MySQL database, can be retrieved by using a request
4. When the PDP evaluates the policy, *organization-name* is missing in the request context because it hasn't been sent by the PEP. Thus, it has to call the PIP that retrieves this attribute from the MySQL database. **It implies the XACML PDP of the cloud provider contains the PIP code that implements MySQL requests and the associated configuration (address of the MySQL server, login and password, the database schema) to get the attribute value.**
5. Then, the PDP evaluates expression "geometry-contains" to check whether the requested map is within MY-ORG-AREA or not. This assumes **the PDP of the cloud provider has the code that implements function "geometry-contains", and data type "geometry"**.
6. Finally, the PDP returns its decision to the PEP.

This simple example raises a deployment problem. The cloud provider must provide its customer non-XACML standard features: the PIP to get the value of attribute organization-name and the code to implement function "geometry-contains". The first issue is how to make these non-standard features reusable and configurable in order to allow *1) the cloud customers to adapt the authorization system according to their authorization policy and 2) the cloud providers to minimize the cost of providing such flexibility*. The second issue is how to manage these reusable and configurable features efficiently? These features are mandatory because the authorization policy requires them. If the policy is changed to a new one that does not include any constraint on attribute organization-name nor function "geometry-contains", these two features are no more required. Thus, *how to manage the relation between the policy and the required features*?

3. SELF-CONTAINED POLICY

In a previous article [6], we have proposed the concept of self-contained policy to cope with this issue. The approach we have chosen consists in the policy includes all the information required by a PDP to execute the policy. This approach facilitates the management of the PDP implementation life cycle. When a policy is loaded, the required code and associated configurations are installed. When the policy is unloaded, the associated code and the configurations are removed from the PDP. We present here the concept of self-contained policy together with deployment architecture. This section improves the work presented in [6] in terms of adaptability.

3.1. Self-contained policy properties

In this section, we recall the properties of a self-contained policy.

Property of self-sufficiency: *a self-contained policy should be self-sufficient.*

A self-contained policy should contain all the required information to be executed by any self-contained policy compliant PDP. The policy should not implicitly require any external code or configuration to be executed by a PDP, except those in the standard.

Property of self-description: *A self-contained policy should provide enough information for managing its life cycle.*

When a self-contained policy is loaded, the self-contained policy should indicate how to use its PIP code, function, etc. When the policy is unloaded, the self-contained policy should provide the set of associated implementation elements that must be uninstalled by the PDP.

Property of dynamicity: *a self-contained policy should be dynamically loadable and unloadable.*

The PDP should be able to load and unload a self-contained policy without requiring to be stopped.

3.2. Service oriented component approach

We have chosen to follow the service oriented component [7] approach that takes advantages of integration and dynamicity from service-oriented architectures, and reusability and dependency management from component-oriented models.

Service-oriented architecture promotes modeling solutions in term of provided services described by contracts. It is based on the idea of composing applications by discovering and invoking available services to accomplish some tasks [8]. The general pattern of service-oriented solutions consists in service provider, service consumer and service registry (Figure 3). Service providers publish services at run time and service consumers request services for a specific contract. Different service providers can offer the same service and the consumer can choose based on the contract. Since the framework allow service consumers to be dynamically notified of new registered services, and unregistered services.

Component oriented programming focuses on making reusable logical blocks of software that implements one or more interfaces. Component-oriented software is then an assembling of components. The notion of interface being very similar to the notion of service interfaces, component-oriented programming has been used to implement services. A service can be implemented by one or more components.

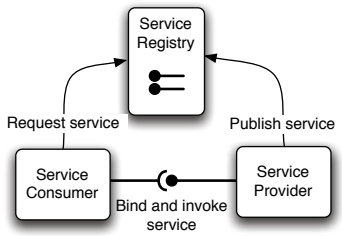

Figure 3. Service-oriented approach

3.3. A service-oriented components based XACML architecture

Our objective being to provide a flexible and reusable authorization service, we have decomposed our architecture into independent entities that can be changed without impacting the whole system. Thus, it is composed of:

- A *core authorization decision engine* that represents the basic functionalities. It consists in the PAP component that controls the life cycle of the policy, the PDP component that provides decisions and the context handler component that handles the request context by providing two services for retrieving and modifying the request context. The core authorization decision is independent from the system to control or the policy to evaluate.
- The *interfaces to access the core authorization engine*. The *administrative console interface* component allows policy administrators to access the PAP. We have implemented two different components to access the PAP via a local console, a web service. But other administration interfaces could be defined. We are working on a web interface. The second entity is the *core authorization decision access point* (CADAP) that allows PEP to send requests and get decisions from the PDP. Like administration consoles, different CADAPs can be proposed to adapt the authorization system to different system/network architectures. We have implemented two CADAPs components: one to access the PDP locally via a UNIX pipe and another via a web service.
- The *self-contained policy* that contains the XACML policy with all non-standard functionalities.

This architecture has been implemented using the OSGi service oriented component specification [9] and more specifically the Apache Felix framework [10]. The PDP component is a modification of the Sun's XACML PDP implementation [11].

A self-contained policy includes three kinds of components: the policy access component, the PIP components and the data type components.

The *policy access component* provides only one service, called *get-policy*. The PDP invokes this service to get the policy rules from the self-contained policy. When the PAP activates the self-contained policy in the authorization engine, service *get-policy* is automatically registered. The PDP is then dynamically notified that it can load the policy file and start to use it for its decision-making process.

PIP components are logical blocks of software dedicated to retrieve attribute values from different sources, i.e. the attributes are not in the request context. They publish their service *get-attribute* that adds in the request context the value of the attribute. The PDP component or any data type or PIP component can invoke this service. One PIP component can be responsible for one or more attributes. However one attribute is handled by only one PIP component. As a consequence, PIP components should indicate the name(s) of the attribute(s) they are in charge of during the registration phase. If necessary, PIP components might have an associated configuration to perform their task (the ip address of the database server, the login and password, etc). The notion of reusability beside the component approach is important here because the same functionality can be shared by different policies like PIP component that implements the common access storage technologies such as MySQL database or LDAP repository.

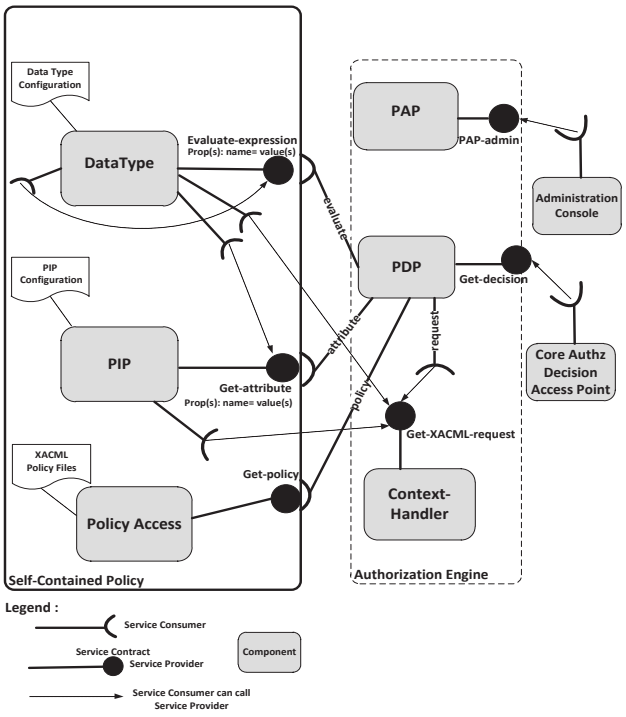

Figure 4. Self-contained policy

Finally, *data type components* implement all the functionalities related to data types elements manipulation. Each data type component provides service *evaluate-expression* to allow the PDP to evaluate non standard expressions in the policy. *Service evaluate-expression* takes as input an expression that uses functions the service handles and returns the result of the expression based on the request context. When a component, including the PDP, needs to evaluate an expression of the form `<Apply FunctionId:"urn:XYZ"> … </Apply>` its sends all the expression to the component responsible for function XYZ to evaluate it. This process can be recursive. If the data type component XYZ find a sub expression `<Apply FunctionId:"urn:ABC"> … </Apply>` it is not able to evaluate, it can ask the corresponding data type component to evaluate it, etc. In addition, data type components can invoke the *get-attribute* service of a PIP component if additional attributes are required. Thus data type components need to indicate the set of function names they handle. Finally, data type components may also require configuration when the policy writer specifies data type values (we refer the reader to [12]).

Self-contained policies are more complex than XACML policy file only. Especially, managing their life cycle is not limited to uploading/removing an XML file only. All the PIP and data types components must be handled too (installing and/or removing from the authorization system when needed – see property of self-description). As a consequence, the PAP component provides the following functionalities:

load a self-contained policy – the PAP loads all the components of a self-contained policy in the authorization engine environment. However, the services provided by the components are not published at this time.

activate a self-contained policy – the PAP activates the components. They publish their services to the registry service. The last activated component is the policy access component. When service get-policy is published, the PDP is then notified that a new policy is ready to use and adds the XACML policy to the list of current policies.

deactivate a self-contained policy - the self-contained policy is still in the environment, but the services are not accessible. When the PDP detects service get-policy of this self-contained policy is no more available, the PDP removes the policy from the list of policies it has to execute.

unload a self-contained policy – the PAP removes all the components of the self-contained policy from the authorization engine.

4. TOWARD XACML AUTHORIZATION AS A SERVICE

Our goal is to propose a an authorization system that at the same time:

1) is flexible/adaptable enough to allow customers to control their resources considering their own specific security requirements and IT infrastructure that could be inside the cloud or elsewhere;
2) minimizes the cost of providing such flexibility to the cloud provider.

We present in this section how to use our service oriented component architecture for achieving this goal.

The first step consists in the initialization of the authorization system (Figure 5.). When the customer asks the provider for an authorization system, the provider grants him access to a generic core authorization engine. To customize this generic authorization system, the cloud provider must provide a list of reusable administration console interfaces (ACI) and core authorization decision access point (CADAP) components. The ACI components represent the interfaces that customers can use to manage policies. Our idea is the provider supplies different ACIs with different features. The basic functionalities of an ACI are load/unload, activate/deactivate a policy. Extra features such as can complete these basic functionalities: activate a policy at a given time, authentication mechanism, code of ACI certified or not, etc. Then the administrator can choose an interface that best fits its requirements. For example, we have developed two administration console interfaces. The first one is a set of commands added to the Felix framework shell [13]. It has been defined to be used a human administrator. The second ACI is a web service that can be accessed a web service client program. Other administration console interfaces can be imagined. The CADAP components represent the connection between the PEP that controls the resource and the authorization system. Different technologies exist for this communication and PEPs might already exist on the customer technology (for example, the grid solution Globus toolkit includes a SAML PEP [14]).

Thus, the cloud provider must provide a list of CADAP with specific features like for ACIs. As an example, we have created a simple web service CADAP.

It should be noted that one customer can chose several ACI/CADAP components for the same decision engine.

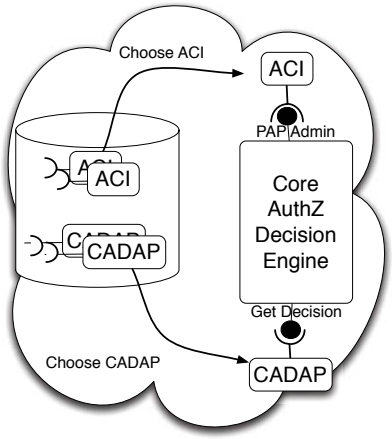

Figure 5. Authorization decision engine initialization

When the initialization phase is done, the customer can write its policies and deploy them on its authorization system.

As we have illustrated in the scenario, XACML policies might require non-standard functions/data types and PIPs. In our example, the administrator wanted to use the functions and data types for manipulating geospatial data defined in GEOXACML and a PIP that can retrieve attribute *organization-name* from his MySQL database.

Like ACIs or CADAPs, the cloud providers supply a set of PIP and data type components. PIP components can differ from the technology to access attributes (MySQL, LDAP, webservice, etc), the quality of the component (performance, code certification, etc). Data type components allow security administrators to take advantage of non-XACML data types and functions for writing complex security policies.

In order to facilitate the task of the administrator, we have implemented a self-contained policy editor based on Netbeans integrated development environment (Figure 6.). Netbeans IDE [15] can be extended by modules to create editors for new languages. Thus the process for editing and deploying a policy is the following:

1) The administrator creates a new project using our editor.
2) When the administrator is editing its policy, he can import PIP or Data Type components. Throughout two services provided by components, the editor can interact with them. The first service is auto-completion to minimize the typographical errors. The second service allows the administrator to graphically configure PIP and Data Type components (e.g. the server address, login/password and SQL statement for a basic MySQL PIP component).
3) When the administrator has finished to edit its policy, the editor creates the self-contained policy file.
4) Finally, the administrator loads its policy in the authorization system using the chosen ACI. At that point, the components in the self-contained policy are installed in the authorization environment. When the administrator activates the policy, all the components of the self-contained policy are activated and the PDP is ready to evaluate the policy.

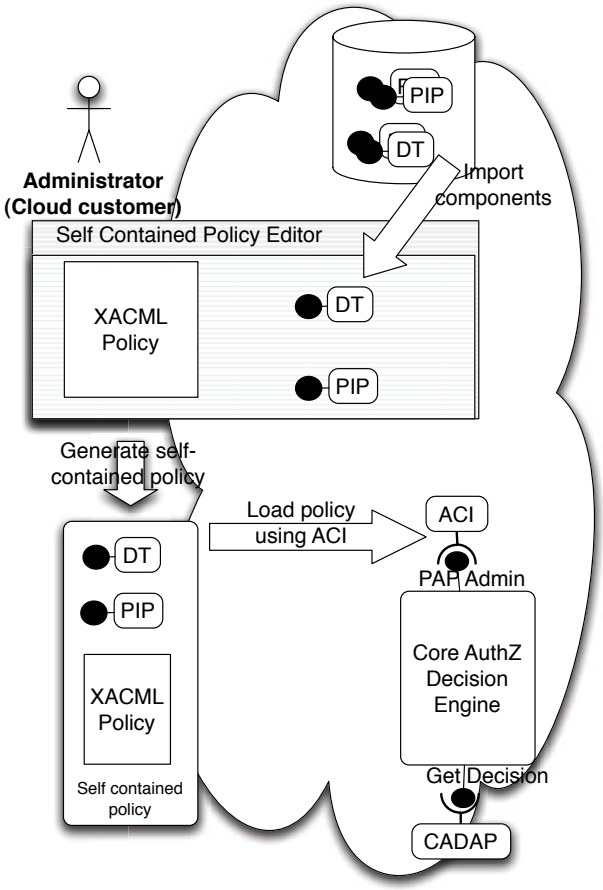

Figure 6. Authorization policy edition and deployment

5. RELATED WORKS

Many implementations of XACML policy decision engine exist. Existing PDP implementations such as SUNXACML [11], Enterprise java XACML [16], XACMLight [17] are complaint to the standard XACML i.e. they support all the standard data types and functions. However, extending these implementations with new features like defining a new data type requires the modification of the source code of the PDP. Finally, HERASAF [18] allows to add dynamically new data type through configuration files. However, this project doesn't define any methodology explaining how to structure data

types efficiently. As a consequence, there is no guarantee a policy can be interpreted even if the implementation of the policy decision engine is dynamically extensible. In addition, there is no management of extension modules life cycle according to the policy, i.e. why and when external modules should be added to or taken away from the policy decision engine.

According to our knowledge, Ulrich Lang has employed the term "Authorization as a Service" in [19] for the first time in a scientific publication. However, his idea of what the provided authorization service is differs from our point of view. He wants to provide model driven security as a cloud service to PaaS users (i.e., SOA application developers). The tool called OpenPMF can generate authorization rules in different format languages based on a policy specified in a domain specific language and the workflow of the SOA application. Thus people don't have to handle security when developing their application in the PaaS. Our goal is not to generate technology specific authorization rules; we want to build an extensible authorization system with reusable functionalities. Our approaches differ because our target users are not developers but security administrators.

6. CONCLUSION

Providing Security as a Service raises new engineering challenges. Security application will have to easily integrate heterogeneous systems (cloud, hybrid and/or classical networks). As a consequence, adaptability of security application is an important issue.

Following the idea of the Cloud Security Association, we have studied how to use XACML for implementing Authorization as a Service. Our solution is based on the service-oriented component programming paradigm. A core XACML decision engine (composed of a context handler, a PDP and a PAP) can be extended by reusable components (ACI, CADAP, PIP and Data Type). This extensibility allows the service provider to meet its customers' requirements and to minimize the cost of such adaptability. Finally, we have explained how to use this architecture for providing Authorization as a Service.

Our architecture has been implemented in OSGi to prove the feasibility. It also has permitted us to gain experience and to improve the architecture. We are following this work by implementing different components to augment the service we can provide. We are also improving our development tools to make the writing of components as well as the edition of policy easier. According to our model, multiple parties can develop components (not only the authorization service provider). We have to enhance our system to consider that components won't have the same quality level (in term of performance, assurance, trust, etc). We have to add mechanisms to prevent a component to impair the global authorization engine process.

ACKNOWLEDGMENT

This work has been partially founded by ITEA2 European project PREDYKOT (Policy REfined DYnamically and Kept On Track – website http://www.itea2-predykot.org/).

REFERENCES

[1] Cloud Security Alliance, "Defined categories of services 2011 version 1.0", Security as a Service Working Group, October 2011.

[2] In Cloud Security Alliance, "Security guidance for critical areas of focus in cloud computing v3.0", November 2011.

[3] OASIS, "eXtensible Access Control Markup Language (XACML) version 2.0", OASIS Standard, February 2005. URL: http://docs.oasis-open.org/xacml/2.0/access_control-xacml-2.0-core-spec-os.

[4] Open Geospatial Consortium, "Geospatial eXtensible Markup Language (GeoXACML)", version 1 corrigendum, OGC 11-017, december 2011.

[5] R. Laborde, M. Kamel, F. Barrere, A. Benzekri, "PEP=Point to Enhance Particularly", in IEEE POLICY 2008, 2-4 June 2008.

[6] M. Cheaito, R. Laborde, F. Barrere, A. Benzekri, "A deployment framework for self-contained policies", in IEEE/IFIP CNSM 2010.

[7] H. Cervantes , R. S. Hall, Autonomous Adaptation to Dynamic Availability Using a Service-Oriented Component Model, Proceedings of the 26th International Conference on Software Engineering, p.614-623, May 23-28, 2004

[8] M. P. Papazoglou ,P. Traverso ,S. Dustdar ,F. Leymann, "Service-Oriented Computing: State of the Art and Research Challenges", Computer, v.40 n.11, p.38-45, November 2007

[9] OSGi, http://www.osgi.org/Main/HomePage, last access June, 2012.

[10] Apache Felix, http://felix.apache.org/site/index.html, last access June, 2012

[11] Sun's XACML implementation, http://sunxacml.sourceforge.net/, last access June, 2012.

[12] M. Cheaito, R. Laborde, F. Barrère, and A. Benzekri, "Configurable Data Types in Policy Based Access Control Management: A Specification and Enforcement Framework", In : SAR-SSI 2010, Menton -France, 18/05/2010-21/05/2010.

[13] Apache Felix Shell, url: http://felix.apache.org/site/apache-felix-shell.html, last access June 2012.

[14] Chadwick DW, Otenko O, Welch V. Using SAML to link the GLOBUS toolkit to the PERMIS authorisation infrastructure. In: Proceedings of eighth annual IFIP TC-6 TC-11 conference on communications and multimedia security, Windermere, UK, 15-18 September 2004. p. 251-61.

[15] Netbeans Platform Learning Trail, url: http://netbeans.org/kb/trails/platform.html, last access June 2012.

[16] Enterprise Java XACML, url: http://code.google.com/p/enterprise-java-xacml/, last access June, 2012.

[17] XACMLight, url: http://sourceforge.net/projects/xacmllight/, last access June, 2012.

[18] HERASAF, url: http://www.herasaf.org/, last access June 2012

[19] Ulrich Lang, "OpenPMF SCaaS: Authorization as a Service for Cloud & SOA Applications" in IEEE CloudCom 2010.