A Systematic Approach To Implementing ABAC

David Brossard
Axiomatics
525 W. Monroe Suite 2310
60661 Chicago
+1 (312) 374-3443
david.brossard@axiomatics.com

Gerry Gebel
Axiomatics
525 W. Monroe Suite 2310
60661 Chicago
+1 (312) 374-3443
ggebel@axiomatics.com

Mark Berg
Axiomatics
525 W. Monroe Suite 2310
60661 Chicago
+1 (312) 374-3443
mark.berg@axiomatics.com

ABSTRACT
In this paper we discuss attribute-based access control (ABAC), and how to proceed with a systematic implementation of ABAC across an enterprise. The paper will cover the different steps needed to be successful.

Keywords
ABAC; authorization; lifecycle; access control; fine-grained authorization; RBAC; process.

1. INTRODUCTION
In Systems & Application Security, authentication has long been the main focus of most standards, implementations, and vendor software. Authorization tends to be overlooked or left to the individual developer to implement in their own applications. In 1992, Ferraiolo et al. published a paper establishing role-based access control (RBAC) as a means to standardize and systematically implement authorization within systems and applications. In RBAC, users are assigned roles and roles assigned permissions. The use of roles and permissions makes access control more manageable. It also provides for a first tangible step towards externalized authorization

However, RBAC also suffers from manageability. With the advent of the Internet, APIs, IoT, Big Data, there is an increasing need for finer-grained, context-aware authorization. RBAC cannot provide access control based on relationships or contextual attributes such as time or location. Consequently, RBAC cannot express real-world access control policies. RBAC simply does not scale to the complexity in today's IT environments. This is where attribute-based access control (ABAC) comes in.

2. ATTRIBUTE BASED ACCESS CONTROL
ABAC is a "next generation" authorization model that provides dynamic, context-aware and risk-intelligent access control. It helps achieve efficient regulatory compliance, effective cloud services, reduced time-to-market for new applications, and a top-down approach to governance through transparency in policy enforcement. ABAC is built on three key elements:

- Attributes,
- Policies, and
- A deployment architecture.

2.1 Attributes
Attributes are the foundation for ABAC. Attributes, put simply, are key-value pairs where the key represents the identifier of the attribute. Attributes can be multi-valued. Attributes are used to describe anyone or anything. In ABAC, attributes are typically grouped in categories. These categories represent the grammatical function the attribute plays:

- The subject category
- The action category (the verb from a grammar standpoint)
- The resource category (the object from a grammar standpoint) and
- The environment or context category.

2.2 Policies
Attributes alone are not enough to express authorization logic. They need to be tied together using policies. ABAC introduces the concept of policies. Policies combine attributes together to express positive and negative cases. These policies are a direct reflection of the authorization requirements application owners have. For instance, in a financial environment, policies might express who is allowed to view, edit, and approve financial transactions. An instance of a positive policy may be:

- A manager can view a transaction in their branch.

An instance of a negative policy may be:

- No one can approve a transaction above their approval limit.

In ABAC, policies can not only be negative or positive, they can also be specific to a particular application, function or transaction or broad enough to apply across an entire enterprise. Policies can be combined together to achieve the relevant authorization scenarios.

ABAC'17, March 24 2017, Scottsdale, AZ, USA.
 ISBN 978-1-4503-4910-9/17/03…$15.00.
DOI: http://dx.doi.org/10.1145/3041048.3041051

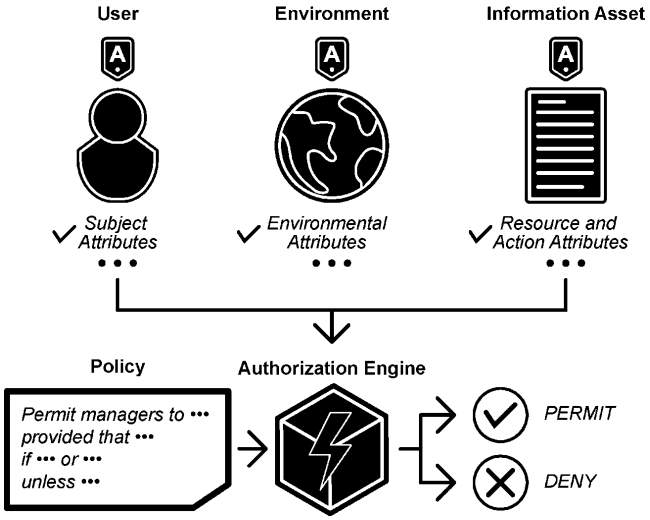

Figure 1. Policies tie attributes together

In order to implement policies in a technical environment, OASIS, the organization for the advancement of structured information standards has been designing a standard called XACML (eXtensible Access Control Markup Language). XACML provides:

- An architecture,
- A policy language, and
- A request-response scheme

2.3 ABAC Architecture

ABAC defines the following architecture and flow:

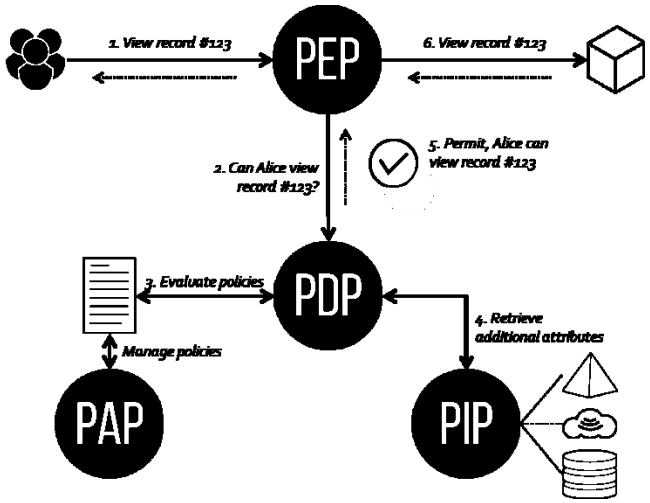

Figure 2. The ABAC Architecture

When an access request is sent from the subject to perform a specific action on the object, it is intercepted by the Policy Enforcement Point (PEP), which transform the business/application request to an authorization request and sends it to the Policy Decision Point (PDP), the authorization decision engine that uses the information provided in the request and the policies to decide whether the request should be allowed or not.

The PDP uses the Policy information Point (PIP) to lookup attributes that are referenced in the policies and hence needed to make a decision on the authorization request.

The Policy Administration Point (PAP), as the name suggests, is the architectural entity that is used to manage policies the PDP later evaluates. It enables the authoring, deployment, and change management of the policies.

3. ABAC CHALLENGES

3.1 A new approach requires new processes

Traditionally, in IAM and RBAC, defining permissions and role assignments belongs to the central IT security team. The team is responsible for defining roles and permissions and assigning them to users. There is a provisioning and de-provisioning process in place. Access reviews are run on a regular basis where user-to-role assignments are checked and signed off by managers.

In ABAC, however, a user's entitlements are not directly assigned to the user via roles and permissions. Rather a user's entitlements are the result of a runtime authorization request evaluated against a set of policies. This means access reviews in the traditional sense are no longer effective. Provisioning and de-provisioning are no longer sufficient.

ABAC will therefore require new processes around provisioning and access reviews. ABAC will also require that new authorization requirements be gathered and implemented.

3.2 Lack of requirements

In RBAC, requirements are either implicit or entirely in the hands of application developers who implement the requirements as code inside their application.

In ABAC, authorization requirements need to be gathered and implemented as centrally-managed authorization policies. This therefore calls for a new set of steps: use case definition and authorization requirements gathering.

These requirements will need to be maintained both centrally and at the level of the use case or application where they stem from. There will be a need for a process around the requirements to define, refine, approve, and implement them. This paper aims to elicit such a process.

3.3 Unclear ownership

Another challenge in ABAC is the ownership. In RBAC, most of the ownership and responsibility lies with the central IAM team. This is because they define coarse-grained access within the RBAC system and let developers implement fine-grained controls in the application. This dilutes responsibilities.

In an ABAC approach, the entire authorization logic is expressed inside the authorization policies. This entails that the central IT team and the application owners / business analysts will need to talk to each other to define the requirements and agree on ownership.

ABAC brings the unique ability to bridge the gap between IT and the business side of any enterprise.

4. IMPLEMENTING ABAC IN THE REAL WORLD

4.1 Overview

We propose the following authorization policy lifecycle as a systematic approach to implementing ABAC within enterprises.

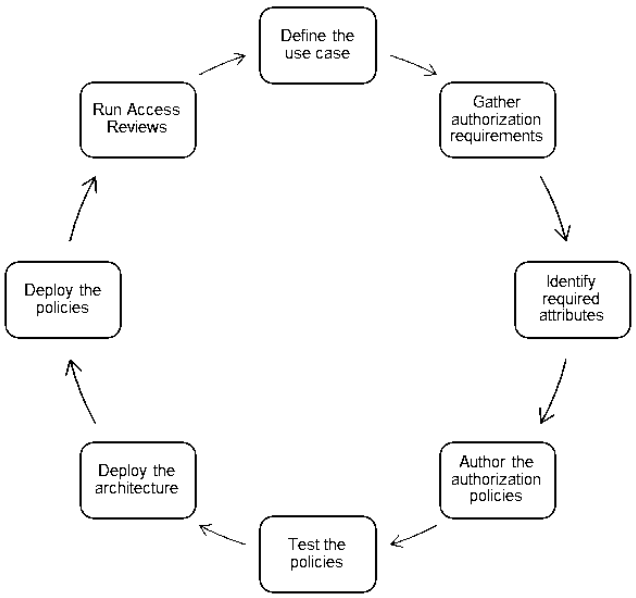

Figure 3. The Authorization Policy Lifecycle

4.2 Actors involved

The following actors will be involved in the process:

- The application owner is responsible for defining the overall use case
- Business analysts, security architects, and security officers (compliance & privacy managers) are responsible for defining authorization requirements for the given use case
- Business analysts, architects, and data owners are responsible for defining the required attributes and their source.
- Application developers or policy authors are responsible for authoring the policies
- Application developers, policy authors, and business analysts are responsible for defining, implement, and running policy tests
- Architects and application owners are responsible for deploying the architecture and the policies.
- Compliance & audit managers and application owners are responsible for running ABAC access reviews.

4.3 Define the use case

This is the first step in an ABAC implementation project. Implementers should use it to discuss and frame the use case with various stakeholders, in particular:

- The application owner,
- The business analyst, and
- The security architect.

The goal of this step is to provide context and an achievable scope. Many projects

4.3.1 Example

Acme Inc. wants to let its users access records easily. This applies to employees, customers, and partners. However at the same time, not all individuals should have access to all records.

4.4 Gather authorization requirements

An authorization requirement is a statement or natural language authorization policy which states what should be allowed or disallowed. Once the use case has been clearly identified, authorization requirements can be authored. When doing so, the priority or precedence of the requirements may not need to be clarified yet.

Authorization requirements may come from several sources and stakeholders. For instance some requirements may relate to how a business operates (e.g. working hours). Others may relate to security best practices (e.g. encrypting data). Others still come from regulations (e.g. privacy regulations or PCI-DSS). Others come from the business owners.

The goal of this step is to gather all the requirements and identify their owner (or stakeholder).

4.4.1 Examples

The following are examples of authorization requirements

- A manager can view any record.
- An employee can view a record in their own department

4.5 Identify required attributes

Once requirements have been listed, it now becomes necessary to identify the attributes used in those requirements. In this stage, read through the natural language statements previously defined and break them down into attributes. Identify each one of them and their origin. For each attribute, one should identify the following parts. The more specific the definition, the better the policy will be. It will also increase the likelihood the policies and requirements will be clearly understood. Lastly, it will facilitate access reviews.

4.5.1 Short Name

An attribute's short name is part of its identifier. It is what most policy authors would use in a policy. For instance, `role`, `department`, and `approvalLimit` are all examples of short names for user attributes. A short name needs to be unique within the context of a given namespace. The short name needs to comply with the ALFA naming convention.

4.5.2 Namespace

An attribute's namespace is the logical domain it belongs to. Policy authors and business analysts can use namespaces to organize attributes into different domains. For instance, Acme may want to use a namespace for customers and a different one for employees:

- `com.acme.user.employee`, and
- `com.acme.user.customer`

A namespace is part of an attribute's unique identifier. The concatenation of a namespace, a "." separator, and the attribute's short name constitutes the attribute unique identifier as well as its XACML identifier should the implementer choose XACML to implement their ABAC policies. An example identifier is `com.acme.user.employee.approvalLimit`.

4.5.3 Category

As previously discussed, in ABAC all attributes fall into different categories. The categories define the grammatical function the attribute plays in the authorization requirement. In ABAC, there are four commonly recognized categories:

- Subject,
- Action,
- Resource, and
- Environment.

XACML also uses these categories and also defines a handful of optional ones. In the light of the previous example, the attribute `com.acme.user.employee.approvalLimit` would belong to the subject category.

ABAC allows for implementers to define their own categories. Experience shows however that using the four aforementioned categories only leads to an easier and more successful project.

4.5.4 Data type

The next step is to define data types for each attribute. When defining the authorization requirement, an attribute's data type is either overlooked or inferred. However, when implementing the requirement into ABAC and then into an actual policy language e.g. ALFA or XACML, it is necessary to specify the data type. The most common data types are:

- String
- Boolean
- Numerical (integer or double)
- Date & time types

In practice, implementers need to look into the data types made available by the chosen framework.

In the examples previously used, we can use the following data types:

- `com.acme.user.employee.approvalLimit`: integer
- `com.acme.user.role`: string

4.5.5 Value constraints

In addition to defining an attribute's data type, providing a value range is also an effective mechanism to providing concise authorization requirements and policies. There are several possible ways value constraints can be specified:

- A list of discrete values, for instance a list of country codes as defined by ISO 1366.
- A pattern e.g. a regular expression (any word starting with A.)
- A range e.g. any letter from A to Z or any number from 0 to 1000

Providing constraints will help policy authors implement policies correctly and avoid typographical mistakes. Constraints will help when running access review tests.

4.5.6 Cardinality

Along with defining a data type and a valid constraint, it is recommended business analysts define a cardinality for each attribute. Knowing and enforcing attribute cardinality will lead to better policies, better access reviews, better authorization requests, and better enforcement.

A cardinality relates to the number of values any given attribute can have at any given time. For instance, `dateOfBirth` is typically an attribute that can have a maximum of one value. The attribute `citizenship`, however, could potentially be multi-valued. Other attributes e.g. `role`, `department`, or `approvalLimit` do not have a clearly defined cardinality. This will depend on the scenarios these attributes are in.

4.5.7 Source

ABAC does not specify where attribute values come from. However, in real-world implementations, attribute values need to be resolved either from a System of Record (SoR) or from the application the ABAC system is protecting. In XACML terminology, the SoR is called a Policy Information Point (PIP) and the application is referred to as the Policy Enforcement Point (PEP). When it comes to attribute value source therefore, the source will either be the PEP or the PIP.

It is important to document where the value comes from. Typically key attributes e.g. a user identity, an action, or an application identity will come from the PEP whereas derived attributes – attributes which values depend on the value of a key attribute – will come from the PIP. The attribute `approvalLimit` is a derived attribute which could be read from a user database PIP based on the key attribute `userId`.

4.5.8 Contact

Lastly, it is important to keep track of the data steward or attribute owner. This individual is responsible for maintaining the integrity, availability, and trustworthiness of the attribute sources. The attribute owner is also responsible for the attribute governance, i.e. how often the value for the given attribute is verified or certified for accuracy.

In ABAC, many challenges have risen from the fact that attributes used in policies were not marked as such and attribute owners did not realize the importance of the information they maintain. These attribute owners must be fully aware of the fact their attributes are being used in ABAC policies in a privilege-giving way.

4.6 Author the authorization policies

Defining attributes is akin to defining the building blocks with which the policies can then be built. A list of clearly-identified attributes will make this step, policy authoring, extremely simple. The aim of this step is to take the natural language statements and work them into machine-interpretable statements.

One of the aims of this step is to eliminate any doubt introduced by natural language. In this example below, what does `own` mean? A machine would not be capable of making the right inference. This is why natural language policies need to be broken into atomic attributes and attribute comparisons. This step also normalizes values. For instance, instead of using "Managers", the normalized value "manager" will be used. Note that the singular, lower-cased value is used. This choice is up to the policy author and business analysts.

This step does not focus on policy combination, hierarchy or precedence. The result of this process is to end up with an implementable list of policies typically written in the order the policies will be evaluated.

The following diagram summarizes the process through which natural language statements are broken into attribute-based policies which can then be directly implemented into ALFA or XACML.

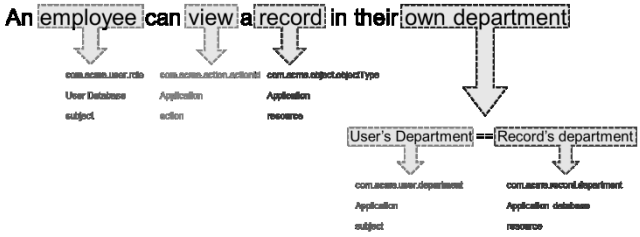

Figure 4. Breaking down natural language policy statements

ALFA and XACML are two languages that are available to implement the policy into machine-interpretable statements.

4.7 Test the policies

Once requirements, attributes, and machine-interpretable policies have been defined, policy authors should agree on acceptance tests and implement them. Defining acceptance tests is part of the authorization requirements step. This step focuses on implementing them.

It is important to put the responsibility to define the tests in the hands of the application owners and the business analysts. Policy authors and application developers are merely responsible for implementing the tests into machine-executable tests.

There are different types of tests that can be run against ABAC policies:

- Binary testing: Can a given user perform a given action on a given resource under a set of given circumstances?
- Gap analysis: Is there any way a given user can perform a given action on a given resource?
- Reverse Query testing: What can a given user do? Who can access a given resource?

Binary testing is the easiest test to put into place and is covered by XACML. In binary testing, authorization requests are defined. Expected responses including a decision are defined. Tests are then run one by one or in batch mode. In each test, the expected response is compared to the actual response.

A comparison may include:

- The decision
- The set of policies considered to reach the decision
- Additional metadata returned e.g. obligations, advice, and attributes.

A policy tester will need to define the scope of the comparison. Binary testing coverage needs to be computed from the policies, the attributes, and the attribute values (the range and cardinality).

A policy tester must implement a test harness that iterates through the potential values and generates the relevant tests to guarantee test coverage.

Tests can then be run every time a policy is added, edited, deleted, and promoted to upper environments.

Gap analysis and reverse query tests are outside the scope of this paper.

4.8 Deploy the architecture

With Attribute-Based Access Control, you can choose where and how to deploy your Policy Enforcement Point. Where you deploy the PEP determines what type of authorization you can achieve and how broad the protection is.

The following diagram illustrates some of the most common layers where PEPs can be implemented. The small box to the left represents the PEP.

The location of the PEP and the interfaces it goes against may impact the access granularity. For instance, integrating with a Web SSO layer will be relatively coarse-grained as the PEP will not have access to any message payload. Integrating with an API gateway inside the API tier will be finer-grained as the PEP will have visibility into the API message on the way in and on the way back.

It is the responsibility of the application developer, application owner, and architect to define the most suitable place for a PEP. Note that there can be multiple PEPs deployed at the same time.

Figure 5. The Any-Depth Architecture

4.9 Deploy the policies

The last step consists in deploying the policies. In an ABAC system, this entails promoting the policies from lower environments to higher environments i.e. from development to test to QA and eventually production.

Policies can be handled as code would be. Policies can be checked out from a development system, committed to a version control system, tagged, and checked out in the next environment. Change and version management should be handled through those tools already in place within the enterprise for source version control.

It is the responsibility of the application owner jointly with the owner of the centralized authorization ABAC service to deploy new versions of the policies.

In an ABAC system, the Policy Decision Point (PDP) is a stateless authorization engine. This means that the PDP can be deployed alongside other PDPs behind a load balancer. All these PDPs therefore act as the same logical PDP. This means that a policy can be progressively deployed to one PDP after another for final testing.

An ABAC solution should have the means to push configuration to an entire set of PDPs in one go. An ABAC solution should also provide the means to rollback policies to a previous version.

4.10 Run access reviews

The last step in the ABAC authorization policy lifecycle is to run access reviews.

4.10.1 Access Reviews in RBAC – Pros & Cons

Traditionally, in an RBAC system, an access review consists of a long list of users and their entitlements i.e. their assigned roles and corresponding permissions.

Such reports can be directly produced from the central RBAC system which statically stores the user → role → permission assignments.

The challenge with such reports is that they are very lengthy. In fact many managers will blindly accept the results of the access review reports for their subordinates because they cannot simply review each entitlement one by one. In addition these reports tend to be very technical with role names and permissions names that no longer bear any authorization semantic but rather are named based on technical constraints e.g. SYS1_ROL_XYZ_FN1. Lastly, these reports are only part of the story: in an RBAC approach, part of the authorization, the relationships, the context, and other fine-grained constraints are all implemented in an application's codebase therefore escaping the access review reports.

4.10.2 Access Reviews in ABAC – Pros & Cons

In an ABAC system, an access review consists of an analysis of the policies against a set of attributes to determine what these attributes grant. For instance, given a set of user attributes (identity, role, department…) what can those users achieve?

Such reports can be produced by evaluating partial requests against an ABAC authorization engine configured with the relevant policies.

The key benefit of this approach is that it shows the direct root "cause" for access to be granted. For instance, given the access review request "what can a manager do?", the report will contain:

- View documents if user.department == doc.department
- Edit documents if user.userId == doc.owner
- Etc.…

In the ABAC authorization policy lifecycle, it is important to run access reviews on a regular basis to determine whether a given user profile (a manager) or a specific user (Alice) have gained or lost entitlements.

5. AN EXAMPLE: IMPLEMENTING AN ABAC USE CASE USING XACML & ALFA

In this section, we run through the process using a simple example.

5.1 Define the use case

Acme Inc. is a company like any other. It has thousands of employees spread across multiple sites, partners with other entities, sells to a wide range of customers and runs many different applications and services. Above all, it has a lot of information records – data if you will.

Acme Inc. wants to let its users access records easily. This applies to employees, customers, and partners. However at the same time, not all individuals should have access to all records.

5.2 Gather authorization requirements

Based on the sample use case aforementioned, here is a list of sample authorization requirements. This list is not exhaustive.

- A manager can view any record.
- An employee can view a record in their own department
- An employee can edit a record they own, if it is in draft mode
- A manager can publish a record if the record is in final mode and it belongs to an employee below that manager.

5.3 Identify required attributes

The following table lists the attributes that are needed to implement the authorization requirements aforementioned.

Table 1. Sample list of attributes

Short name	Namespace	Category	Data type	Value range
role	com.acme.user	subject	String	employee, manager
actionId	com.acme.action	action	String	view, edit, publish
objectType	com.acme.object	resource	String	record
status	com.acme.record	resource	String	draft, final
department	com.acme.user	subject	String	sales, finance, engineering
department	com.acme.record	resource	String	sales, finance, engineering
owner	com.acme.record	resource	String	Alice, Bob, Carol...
employeeId	com.acme.user	subject	String	Alice, Bob, Carol...
subordinate	com.acme.user	subject	String	Alice, Bob, Carol...
recordId	com.acme.record	resource	String	123, 456, 789...

5.4 Author the authorization policies

In this step, we rework the authorization requirements (natural language statements) into machine-interpretable statements.

- Original rule: An employee can view a record in their own department.
- Reworked rule: a user with the role == "employee" can do the action == "view" or action == "edit" on resources of type == "record" if and only if user.department==record.department.

The ambiguity of the natural language has now disappeared. The policy can be easily rewritten using ALFA.

ALFA is the Abbreviated Language For Authorization. It is a lightweight notation for authorization policies. It maps directly into XACML. Its aim is to make policy authoring much simpler for developers than native XACML allows.

```
policy records{
    target clause object.objectType == "record"
    apply firstApplicable
    /**
     * R2 - An employee can view a record in their own department
     */
    rule employeesView{
        target clause user.role == "employee" and action.actionId == "view"
        condition user.department == record.department
        permit
    }
}
```

5.5 Test the policies

In this step, we define and implement the acceptance tests. The following are good examples of acceptances for the policy defined previously:

- Can an employee in sales view a record in sales?
- Can an employee delete a record?
- Can an employee in sales view a record in finance?
- Can an employee view a record?

These tests can be implemented using XACML or the JSON profile of XACML.

5.6 Deploy the architecture

In this step, the architect needs to determine where the integration will take place. In this example, one can imagine the record management system is exposed over a REST API. In this case a natural integration for the PEP would be on the REST API layer or inside a REST API gateway.

5.7 Deploy the policies

In this step, the policies are simply deployed to the PDP.

5.8 Run Access Reviews

The last step in the authorization policy lifecycle is to run access reviews. Given the example, sample access review requests include:

- What can Alice do?
- What can a manager do?
- What can an employee do?
- What can an employee do?

6. CONCLUSION

This paper introduced a systematic approach to ABAC which enterprises can follow to ease the adoption of ABAC within their IT landscape. Processes change between RBAC and ABAC systems. Some processes no longer exist. Many new processes are required. The authorization policy lifecycle aims at simplifying the migration and adoption project. It has been designed over the years based on dozens of customer implementations.

7. ACKNOWLEDGMENTS

We would like to express our thanks to the entire Axiomatics team for helping us with the paper and proof-reading it. Our particular thanks go to Babak Sadighi, Erik Rissanen, Ludwig Seitz, Andres Martinelli, Finn Frisch, Andreas Sjöholm, Pablo Giambiagi and Srijith Nair.

8. REFERENCES

[1] Ferraiolo, D. and Kuhn, R 1992. Role-Based Access Controls. *15th National Computer Security Conference*.

[2] Hu, V. et al., Guide to Attribute Based Access Control (ABAC) Definition and Considerations, NIST Special Publication 800-162, Nat'l Institute of Standards and Technology, Jan. 2014; http://nvlpubs.nist.gov/nistpubs /specialpublications/NIST.sp.800 -162.pdf.

[3] Rissanen, E. et al., eXtensible Access Control Markup Language (XACML) Version 3.0, *OASIS eXtensible Access Control Markup Language (XACML) TC*, http://docs.oasis-open.org/xacml/3.0/xacml-3.0-core-spec-en.html

[4] Giambiagi, P., Nair, S., and Brossard, D., Abbreviated Language for Authorization Version 1.0, *OASIS eXtensible Access Control Markup Language (XACML) TC* https://www.oasis-open.org/committees/download.php/55228/alfa-for-xacml-v1.0-wd01.doc

[5] Rissanen, E. and Sadighi, B., Access-Control Policy Administration in XACML, *13 October 2005, CRCIM News No. 63.* Available at http://fmt.isti.cnr.it/WEBPAPER/p38-39.pdf.

[6] Glazer, I., Nailing Down the Definition of "Entitlement Management", *Gartner Blog Network*. Available at http://blogs.gartner.com/ian-glazer/2009/05/13/nailing-down-the-definition-of-entitlement-management/

[7] Giambiagi, P. et al., A system and method for evaluating a reverse query, US Patent PCT/SE2011/050955

[8] Brossard, D et al., JSON Profile of XACML 3.0 Version 1.0, *OASIS eXtensible Access Control Markup Language (XACML) TC* http://docs.oasis-open.org/xacml/xacml-json-http/v1.0/xacml-json-http-v1.0.html

[9] ISO 3166-1 Country Codes, International Organization for Standardization, http://www.iso.org/iso/home/standards/country_codes.htm