Efficient Attribute Based Access Control for RESTful Services

Marc Hüffmeyer and Ulf Schreier

Furtwangen University of Applied Sciences, Germany

Abstract. The popularity of REST grows more and more and so does the need for fine-grained access control for RESTful services. Attribute Based Access Control (ABAC) is a very generic concept that covers multiple different access control mechanism. XACML is an implementation of ABAC based on XML and is established as a standard mechanism. Its flexibility opens the opportunity to specify detailed security policies. But on the other hand it has some drawbacks regarding maintenance and performance when the complexity of security policies grows. Long processing times for authorization requests are the consequence in environments that require fine-grained access control. We describe how to design a security policy in a resource oriented environment so that its drawbacks are minimized. The results are faster processing times for access requests and an easy to manage concept for security policies for RESTful services.

1 Introduction

Many of today's information systems and applications manage huge amounts of users and data. Often users share their own content (e.g. photos, documents) within these applications. A substantial need to control who may access this content is the consequence. In an environment where a lot of users share a lot of data and specify multiple access rights, a flexible, high-performance access control mechanism is required. Because older access control mechanisms do not fit the need for flexibility, research is required how to implement newer and flexible access control mechanism so that high performance can be guaranteed even in complex environments. Processing times in general should be kept small to provide a excellent user experience and therefore processing times for access requests in particular must be kept small.

Attribute Based Access Control (ABAC) may be the next important concept in access control [11]. The main idea behind it is that any property of an entity can be used to determine authorization decisions. The eXtensible Access Control Markup Language (XACML) is a standard that describes how to implement attribute based access control [9]. It consists of three parts: an **architecture** describes multiple components and their responsibilities in the authorization context, a **declaration language** can be used to specify access control policies based on XML and a **request language** to formulate access requests and responses. This work focuses on the declaration language. There are three core elements in the structure of a XACML document: **Rules** describe if an access

T. S. Heinze, T. M. Prinz (Eds.): Services and their Composition, 7th Central European Workshop, ZEUS 2015, Jena, Germany, 19-20 February 2015, Proceedings – published at http://ceur-ws.org

request is permitted or denied. **Policies** group different rules together and **policy sets** group different policies together. Policy sets may contain also other policy sets enforcing a hierarchical composition. Each of these elements has a **target** that describes if the element can be applied to a request by defining a condition. A single access request may be applied to multiple policy sets, policies and rules. In that case those rules may have different **effects** (*Permit* or *Deny*) and a winning rule must be found (based on the structure of the policy). XACML uses combining algorithms for that purpose. An example for a combining algorithm is *PermitOverrides*. It states that an applicable rule with the effect *Permit* will always win against a rule with the effect *Deny*. A full list of algorithms can be found in [9].

Listing 1 shows a simplified version of a XACML policy. For simplification some required information like data types or matching methods (e.g. equals, greater than) are removed. The policy contains two rules and is applicable to a HTTP *GET* request on a resource */users/1/photos*. The first rule grants access to a user identified by an URI */users/2* and the second rule prohibits access for any subject. As one can see fine-grained policies may easily become very complex and performance and maintenance need to be optimized.

```
<Policy CombAlg="FirstApplicable">
  <Target>
    <Resource designator="URI" value="/users/1/photos" />
    <Action designator="HTTP-method" value="GET" />
  </Target>
  <Rule Effect="Permit">
    <Target>
      <Subject designator="URI" value="/users/2" />
    </Target>
  </Rule>
  <Rule Effect="Deny"/>
</Policy>
```

Listing 1. Simplified XACML granting GET access for one user to the photos of another user

2 Related work

XACML computes access decisions at runtime and must evaluate multiple attributes of different categories to find a decision. Therefore the average computation time for an access request increases with growing policy complexity. The problem of computation at runtime is related to the architecture resp. the general concept of XACML. A graph based approach described in [10] tries to address performance issues by changing the processing algorithms. Two different trees are used to evaluate an access request. The first tree identifies applicable rules. The second tree holds the original structure of the security policy and identifies the winning rule. Another approach uses numericalization and normalization to optimize performance [4,5]. Numericalization converts every attribute to an integer

value. Normalization converts every combining algorithm into FirstApplicable. In [7] processing time is optimized by reordering policy sets and policies based on statistics of past results. A similar approach to ours also reorders policies based on cost functions but focuses on categories rather than attributes [8]. Also they assume that a rule always is a 4-tuple of a subject, an action, a resource and an effect. Other categories and combinations are not allowed. Declarative authorization for RESTful services is handled in [3]. Attributes are not considered in this approach. In [13] an architecture is described to secure web services (SOAP) based on attributes.

A second major problem of XACML is the modification of policies. XACML does not define how to handle changes to a security policy. The most common way is manually inserting new policy sets, policies and rules supported by a graphical user interface like in [6]. But manually modifying complex policies is very error prone because multiple changes in different branches of the structure may be required. A lot of works exists that addresses the manipulation of XML documents [1,12]. But in this work the security context of XACML is not taken into account. Therefore changes of the security policy are hard to handle.

3 Efficient policy design

An efficient security policy design should enable fast request processing and should be easy to maintain. The security policy described in XACML is a unidirectional graph without cycles. To enable fast request processing we need to consider the costs of processing an access request in a single node of that graph. We define the cost function as:

$$c : N \times Q \to \mathbb{Q}$$

where N is the set of nodes in the security graph, Q is the set of possible access requests and $\mathbb{Q}$ is the set of rational numbers. The set of child nodes of a node k can be expressed as:

$$S_k := \{s \in N \mid \exists p \in path(k, s) \ \& \ length(p) = 1\}$$

Let α_k be the combining algorithm of a node k and let A be the set of combining algorithms. Let ϵ be an effect within the set of effects E. Then one has:

$$\forall \alpha_k \in A \ \exists \epsilon \in E : compute(s_i) = \epsilon \Rightarrow \alpha_k \ stops; \ s_i \in S_k$$

That means that for any given combining algorithm there are one or more effects that cause the algorithm to stop if one of its child nodes computes to one of these effects. For example a node may half two child nodes and the combining algorithm *PermitOverrides*. If the first child node computes to *Permit* the effect of the node will also be *Permit* no matter what the result of the second child node may be. Therefore the combining algorithm stops and should not process the second child node. We define a function γ that describes this behavior:

$$\gamma_k(q, s_i) = \begin{cases} 1 & \text{if } \forall s \in \{s_1, ..., s_{i-1}\} : \alpha_k \ does \ not \ stop \\ 0 & \text{if } \exists s \in \{s_1, ..., s_{i-1}\} : \alpha_k \ does \ stop \end{cases}$$

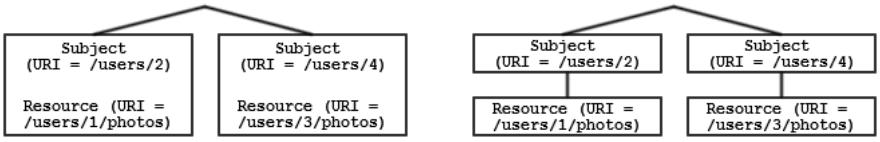
With q ∈ Q and i ∈ $\{1, ..., | S_k |\}$. The cost function c then can be expressed as:

$$c_k(q) = t_k(q) + \sum_{i=1}^{|S_k|} \gamma_k(q, s_i) * c_{s_i}(q)$$

The function $t_k(q)$ describes if a target matches the request q. Therefore it is mainly dependent on how many attributes are specified in the target. Hence, the costs for processing a node depends on the number of attributes in the target, the sum of child nodes and the combining algorithm resp. the order of the children. The following sections describe how to minimize the costs of processing access requests and decrease maintenance efforts for each of the listed factors.

3.1 Target design (minimize t_k)

Attributes should be added carefully to targets to keep the target small and thus reduce the number of comparisons needed to be executed in the worst case. For example a security policy may contain two conditions. Each condition specifies a subject attr. (URI = /users/*userid*) and a resource attr. (URI = /users/*userid*/photos). An intuitive way would be handling each condition in a single target of a rule as indicated in Fig. 1. (a). Processing a request with a subject attr. (URI = /users/2) and a resource attr. (URI = /users/3/photos) requires four attribute comparisons in the worst case because XACML does not specify an order in which attributes must be checked. If a single condition is splitted into multiple targets of rules, policies and policy sets as indicated in Fig. 1. (b) a max of three comparisons is required. This optimization has a benefit for targets that are not applicable to a request while the cost for an applicable rule remains unchanged. Variations in processing times are reduced to a minimum. The maintenance benefit is that it becomes easier to add new conditions that affect a subject with attr. (URI = /users/*userid*) but not a resource with attr. (URI = /users/*userid*/photos).

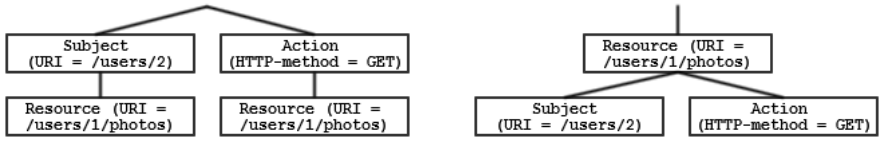


(a) Max. of 4 comparisons (b) Max. of 3 comparisons

Fig. 1. Target design

3.2 Number of sub nodes (minimize $\sum_{i=1}^{|S_k|} c_{s_i}$)

It is obvious that the overall processing time for few nodes is less than the overall processing time for many nodes. Hence, wherever possible targets should be grouped together. That means an efficient policy design must have its branching points at the lowest possible position. Besides the performance gain maintenance efforts for the resource with attr. (URI = /users/1/photos) are reduced because it does not occur twice (cp. Fig. 2.).



(a) Upper branching point (b) Lower branching point

Fig. 2. Number of sub nodes

3.3 Combining algorithm and node order (minimize γ_k)

The selection of the combining algorithm and the child node order also has an effect on performance. Processing those rules first that override the effects of others leads to shorter average processing times. If an overriding rule matches, no other rule needs to be checked. And if no overriding rule matches, the combining algorithm can stop after the first match of the non-overriding rules because they cannot be overridden. This is the basic idea of normalization [5]. Fig. 3 shows the effect of normalization. A given policy with the combining algorithm *DenyOverrides* and two rules as indicated in Fig. 3(a) is transformed so that is has a combing algorithm of *FirstApplicable* and a node order that gives performance improvements. In Fig. 3 (a) both Rule A and Rule B must be processed to find a decision. But for the policy in Fig. 3 (b) it might be enough to process Rule B.

(a) Not normalized (b) Normalized

Fig. 3. Normalization

4 XACML for RESTful Services

One core concept of REST is resource orientation [2]. Therefore we also want to build the security policy based on resources. This is a reasonable technique in a resource oriented architecture and offers the benefit of very fast identification of authorization rules that must be applied during the evaluation process. Therefore targets of policy sets must only contain one resource attribute: the URI. With this constraint it is not necessary to consider combining algorithms since multiple matches of different policy sets or policies are not possible. That means that *FirstApplicable* can be used in every policy set to improve performance. In consequence the processing time for access requests is nearly constant even if new resource paths are added or the security policy is extended.

Another important concept of REST is an uniform interface. Therefore we consider that the set of allowed methods is limited to the HTTP methods. We use these methods as possible actions in a security policy. For each action a new policy should be used right under the policy set for the requested resource. Within these policies rules may be specified that describe under which circumstances the resource may be accessed.

Figure 4 shows an efficient security policy for a RESTful application that follows the optimizations described in the previous sections.

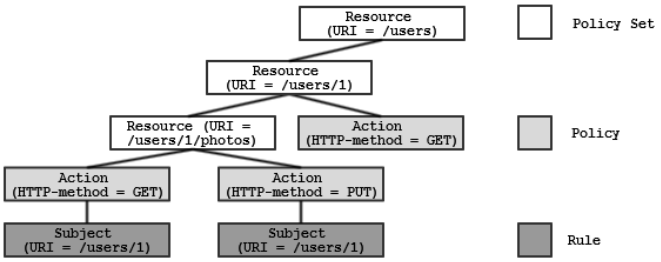


Fig. 4. An example of efficient XACML for RESTful services

5 Results

We performed some first, simple and fragmentary tests on different security policies designed to protect a RESTful service. A first set of policies contains 10 conditions on the URI attribute of 10 resources. For each of the main HTTP methods (GET, POST, PUT, DELETE) we assigned a single policy with one rule, resulting in 40 rules per policy. A second and third set have 100 resp. 1000 more resources resulting in 440 resp. 4440 rules per policy. Each set contains 4 policies: a non-optimized policy (flat structure with a lot of rules in the same policy that follows the pattern of Fig. 1 (a)), a normalized policy with the

optimizations described in section 3.3, a structure optimized policy containing the optimizations described in section 3.1 and section 3.2 and finally a policy with all of the optimizations described in section 3 that follows the guideline described in section 4. All policies within a single set are functionally equivalent. We used Balana as XACML engine (https://github.com/wso2/balana). The measurement was executed on a dual core system (Intel i7-3250M, 2,90GHz) with 8GB working memory reserved for the tests. Each test was executed 20 times.

Figure 5 shows the average processing time for an access request. As one can see the processing times for the set with the smallest policies only differ insignificantly. But with growing policy complexity the difference becomes considerably. While the average processing time for the optimized policy remains approximately constant at about 15ms, the average processing time for the non-optimized policies increases up to 304ms. The main contribution to the performance benefit of the optimized policy is delivered by the structure changes as indicated by Fig. 5. Normalization only has a significant impact for larger policies with many rules and without an optimized structure and causes great variations in processing time of up to 200% of the average processing time. The optimized, structure optimized and non-optimized policies have a variation in processing time of about 25%.

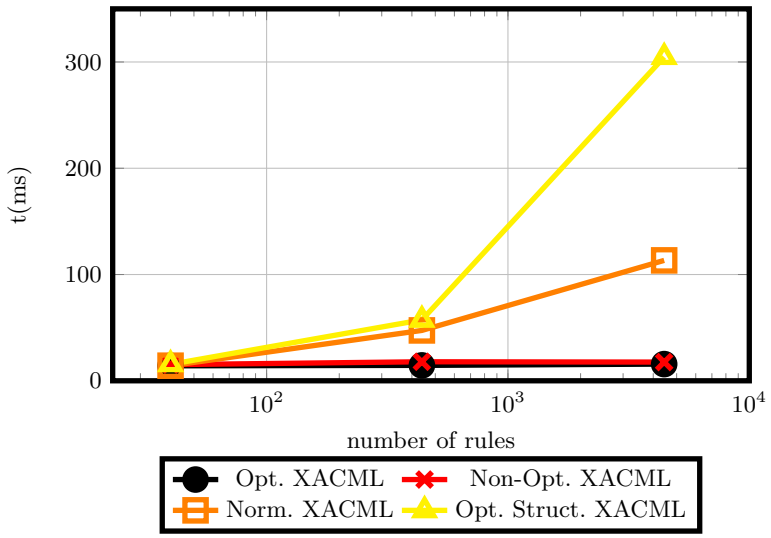


Fig. 5. Average processing time

6 Future work

With every target on a path to a rule access conditions become more restrictive in XACML. This can be a problem for RESTful services. We may have a resource user

list *http://example.org/users* and access to this resource is granted only to some administrators but not to single users. But a resource *http://example.org/users/1* may be accessed by administrators and user 1. Since user 1 is a sub resource of the user list, the policy or policy set that handles access to this sub resource should be placed below the policy set that handles access to the user list. In XACML you cannot extend a condition at sub policy level. In consequence the same condition must be repeated multiple times which causes the policy complexity to grow unnecessarily and increases the maintenance efforts.

To handle the maintenance, performance and restriction problems described in the previous sections, we are developing an alternative language similar to XACML. The language targets RESTful services and should guarantuee that only optimized security policies can be written. A draft version already exists and a prototype is implemented. First results show slightly improved performance even to optimized XACML policies. We want to address the maintenance problems with a structured query language that makes it easy to handle changes in a resource oriented context.

References

1. Cubera, D., Epstein, A.: Fast Difference and Update of XML Documents. XTech '99 (1999)
2. Fielding, T.R.: Architectural Styles and the Design of Network-based Software Architectures. University of California, Irvine (2000)
3. Graf, S., Zholudev, V., Lewandowski, L., Waldvogel, M.: Hecate, Managing Authorization with RESTful XML. WS-REST '11 (2011)
4. Liu, A., Chen, F., Hwang, J., Xie, T.: Xengine: A Fast and Scalable XACML Policy Evaluation Engines. SIGMETRICS '08 (2008)
5. Liu, A., Chen, F., Hwang, J., Xie, T.: Designing Fast and Scalable XACML Policy Evaluation Engines. IEEE Transactions on Compters, Vol. 60 (2011)
6. Lorch, M., Kafura, D., Shah, S.: An XACML-based Policy Management and Authorization Service for Globus Resources. GRID '03 (2003)
7. Marouf, F., Shehab, M., Squicciarini, A., Sundareswaran, S.: Adaptive Reordering and Clustering-Based Framework for Efficient XACML Policy Evaluation. IEEE Transactions on Services Computing, Vol 4 (2010)
8. Miseldine, P.: Automated XACML Policy Reconfiguration for Evaluation Optimisation. SESS '08 (2008)
9. Organization for the Advancement of Structured Information Standard: eXtensible Access Control Markup Language (XACML) Version 3.0. OASIS Standard (2013)
10. Ros, S., Lischka, M., Marmol, F.: Graph-Based XACML Evaluation. SACMAT '12 (2012)
11. Sandhu, D.: The authorization leap from rights to attributes: maturation or chaos? SACMAT '12 (2012)
12. Wang, Y., DeWitt, D., Cai, J.: X-diff: An Effective Change Detection Algorithm for XML Documents. ICDE '03 (2003)
13. Yuan, E., Tong, J.: Attributed based access control (ABAC) for Web services. ICWS 2005 IEEE International Conference on Web Services (2005)