Enhancing Access Control Trees for Cloud Computing

Neil Ayeb, Francesco Di Cerbo(✉), and Slim Trabelsi

Security Research, SAP Labs France,
805, Av. du Docteur Maurice Donat, 06250 Mougins, France
francesco.di.cerbo@sap.com

Abstract. In their different facets and flavours, cloud services are known for their performance and scalability in the number of users and resources. Cloud computing therefore needs security mechanisms that have the same characteristics. The Access Control Tree (ACT) is an authorization mechanism proposed for cloud services due to its performances and scalability in the number of resources and users. After an initial set-up phase, the ACT permits to simplify the evaluation of an authorization request to a simple visit to the tree structure. Our contribution extends ACT towards instance-based access control models by allowing the expression and evaluation of conditions in access control decisions. We evaluated our contribution against an Open Source authorization mechanism to evaluate its performance and suitability to production settings. Early results seem encouraging with this respect.

Keywords: Access control · Data structures · Cloud

1 Introduction

In all its different facets and variations, cloud computing offers a common set of characteristics like elastic computing and resource pooling [4] in order to serve its tenants and their end-users with high performances. To achieve this objective, cloud service providers commit significant resources, for example in terms of processing and storing. Compelled by normative requirements and service level agreements, cloud services must be secured and protected, as well as any data they host; leaks or abuses may result in severe losses for cloud providers, in terms of liability, fines and reputation on the market.

These simple observations lead us to consider the importance of security aspects in the design, implementation and operation of cloud services. It is possible to derive some basic requirements for security controls in cloud computing, among them:

- **R1** cloud security mechanisms must cope with high volumes of requests and with their quick evaluations, and
- **R2** cloud security mechanisms must be able to cope with high amount of users and resources.

S. Casteleyn et al. (Eds.): ICWE 2016 Workshops, LNCS 9881, pp. 29–38, 2016.
DOI: 10.1007/978-3-319-46963-8_3

We claim that these requirements are particularly important for cloud Access Control (AC) mechanisms: they regulate access to cloud resources and operations, by making authorization decisions depending on parameters like the callee's identity, the requested resource or service and the attempted operation. These parameters and the indications on how to analyse them, compose an AC rule.

These rules can be aggregated in two different forms, through AC lists or policies (from now on ACL). The former approach consists of lists of rules that in essence provide an answer to the question, whether a given user could or not access to a given resource. This solution may result limiting, as potentially many combinations of cloud-hosted resources and end-users for each tenant must be explicitly mentioned in ACL, resulting in complex ACL authoring and maintenance. On the contrary, AC policies can capture rules that specify conditions/actions and offer a powerful, yet conceptual, tool of expressiveness and are particularly useful for complex systems [5]. However, in this case it is required a more sophisticated rule processing, performed through a specific AC engine, in charge of rules interpretation. Ultimately, it must be able to make a reasoning process on the given rules and make a decision: permission or denial. Intuitively, a drawback of expressing sophisticated rules is the need for a reasoning process that could be time consuming. In fact, this has been pointed out in [8,9], namely for resource access in cloud computing, where performance could be seriously affected.

In [9], the authors proposed a new access control mechanism to make fast authorization decisions, based on an high-speed caching tree. The Access Control Tree (ACT) is designed in order to simplify the AC decision making the process to a visit to the ACT. Our contribution consists of an extension to this concept to support a form of instance-based AC, by introducing conditions expressed in AC rules as elements of the ACT. This extension permits to represent more sophisticated policies as part of the decision tree, thus easing the adoption of ACT in more complex scenarios. To illustrate our contribution, Sect. 2 presents a number of findings from previous research initiatives to support the need for cloud-specific AC mechanisms, as well as two brief introductions to XACML and to the ACT. Section 3 presents our extension to ACT while Sect. 4 the results of our performance tests. Lastly, Sect. 5 concludes the paper.

2 State of the Art

2.1 Access Control Mechanisms for Cloud Computing

Many popular cloud services use AC solutions that are not specifically designed for the cloud and for its requirements, especially for **R1** and **R2**. This observation is shared by a number of authors (for instance, [7]). This results in a serious gap that can affect at least the configuration and operation of cloud computing solutions. For this reason, Younis et al. [10] identify performance and scalability as the first requirement for modern cloud-ready AC mechanisms.

Well known cloud services like Amazon S3 or Microsoft Windows Azure Storage, still offer very simple ACL-based solutions, essentially allowing access only to known users or publicly available, without giving the possibility to selectively grant access to principals outside of their domain.

The adoption of caching solutions for cloud mechanisms has been proposed in the past. Reeja [6] propose the usage of two policy decision points (PDP), one for new access requests and another one for access requests that were already processed by the primary PDP. Harnik et al. [3] propose a capability-based system that allows the integration of existing AC solutions thus leading to hybrid architectures for AC systems. Such hybrid systems combine the benefits of capability-based models with other commonly used mechanisms such as ACLs or RBAC. On the other hand this study points out the weakness of ACL used by many major cloud providers like Amazon and Microsoft.

2.2 XACML

XACML [2] is an open standard for the specification of access control policies defined by OASIS.

The standard defines a reference architecture for the implementation of the authorization functionality in an application. Such architecture is composed by a number of components. Among them, we find the Policy Enforcement Point (PEP), a part of the application that intercepts and regulates the access to protected resources or functionalities. The PEP permits an access request to take place if authorized by another component, the Policy Decision Point (PDP); the latter is responsible for making the decisions on the basis of a set of security policies and on the parameters of the request transmitted by the PEP (e.g. the target resource, the desired operation, the entity requesting for it). Such decision-making process considers attributes of the user, of the resource, of the system environment. These attributes are used for evaluating conditions stated in security policies expressed using the XACML policy language; the latter is an XML-based standard designed to support ABAC (Attribute Based Access Control). Policies are stored, maintained and made available to the PDP though another architectural component, the Policy Administration Point (PAP), and are evaluated with attributes coming from different sources (e.g. an LDAP directory containing user's attributes) by means of the Policy Information Point (PIP).

In the XACML policy specification, the policy element can be a policy set, a policy or a rule. A policy set is composed of multiple policies, while a policy consists of a number of rules. Each policy has a Target, that defines with which attributes the policy is applicable (e.g., resource, subject, action). Moreover, the policy contains an 'effect' element that determines if the set of identified attributes are related to a Permit or a Deny. It is also possible to define conditions that will be applied prior returning the decision taken by the engine. The XACML standard (with a particular attention in version 3.0) defines 'obligation' handling. An Obligation is an action that must be executed by the PEP when it receives the decision (e.g., logging the access history for a resource). Figure 1 presents an elaboration of the architecture proposed in [9] for the deployment

of XACML AC mechanisms in the cloud. In this setting, the access to cloud resources, be them services, virtual machines, storage or any other, is mediated by the XACML engine that can make decisions considering aspects like multi-tenancy and service-level agreements.

2.3 Access Control Tree

The ACT was introduced by Trabelsi et al. in [9]. The ACT aggregates different policies and their rules in a tree entity in order to make authorization decisions with high performances, through the application of hashing techniques on the tree for efficient data search functions. ACT can be used in an XACML architecture as in Fig. 1 for optimizing the decision making process. For mapping XACML rules into tree elements, Trabelsi et al. adapted a model proposed by Gabillon et al. [1] and adapted it to the XACML policy schema. In their model, only accessible data objects are part of the tree. If the access to an object is denied, it will not appear. This permits to simplify the decision making process and to reduce the number of tree elements. Due to its construction, the tree structure is called 'Permit Tree' and it is indicated in the following with 'P'.

The ACT is composed of four different levels. The first contains the list of authorized subjects (or users, or roles, etc.) declared in the XACML policy repository. The ANY subject ID is used for objects that are accessible to all users with no restrictions. The second level represents the different actions or operations that can be executed on the data. If the list of actions is undefined, there is also an element Any (Action). The third level represents the different types for data objects. Subsequently, the fourth layer contains a list of accessible data object IDs. The layer order (in the example subject, action and resource) can change according to system requirements. For example the first level can be the ID of the object. In that case the selection is made on the object to be accessed, for which one gets the list of authorized users.

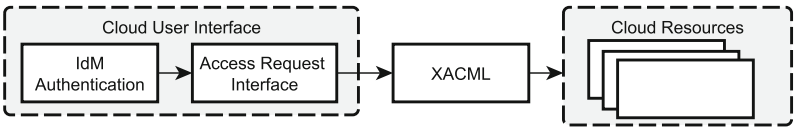

Fig. 1. Architecture for XACML mechanisms in the cloud, adapted from [9]

3 An Extension to ACT

The ACT as presented in [9] consists of a methodology to organise a set of AC directives in the form of a tree structure, in such a way that the evaluation of AC requests boils down to a search navigation in the tree.

The ACT is created by means of an insertion algorithm and queried via a request algorithm. The insertion algorithm allows to construct the tree, structured in four levels: subjects, actions, resource types, resources. The request permits to consider the elements of an AC request and to use them as parameters to search the AC tree. If a result is found, the request can be permitted (in a Permit tree, denied in a Deny tree).

We observed that the ACT achieves the objective of meeting scalability in the number of users and resources together with significant performances. We are focussing this contribution on instance-based Access Control, i.e., methods that permit the expression of fine-grained directives for each individual resource that is under control.

Our proposal has the objective to extend this result further by supporting policies with more sophisticated rules. Therefore, our contribution consists of adding a fifth level to the tree, in order to express the conditions that have to be evaluated at the moment of the decision making process. Multiple conditions may be part of this last layer. A diagram representing the new ACT is depicted in Fig. 2. This new level requires a modification to the ACT insertion and request algorithms, that are used respectively to populate and to query the Permit Tree. A detailed explanation of the algorithms is available in [9].

The new insertion algorithm (that takes advantage of Procedure InsCond) is rendered in Algorithm 1. It starts by considering a set of AC rules in the form $< subject, action, resource >$. If a rule does not include one or more of the tuple constituents, the missing elements are replaced by the special value ÄNY·˙. The algorithm parses each rule and populates the tree by creating a new branch (if necessary) for each touple elements in the respective level.

Our extension considers, for each rule being analysed, whether the rule contains any conditions and in case, they are added to the fifth level of the Permit Tree if they are not already inserted.

Conversely, the request algorithm was modified in order to return the condition(s) that have to be evaluated before issuing an access permission. Algorithm 2 describes this extension.

Procedure InsCond(rule, location)

input : policy rule as R, an ACT node as *Location*
foreach *condition C in R* **do**
  if *C is not in Location* **then**
    Add *C* to *Location*

4 Evaluation and Preliminary Results

We evaluated our proposal against a standard Open Source AC solution, Balana[1], a well-known XACML engine. We compared the performances of our extended

[1] https://github.com/wso2/balana.

ACT and Balana by evaluating a set of up to 3600 policies, generated as the set of all combinations for 60 subjects and 60 resources, performing 7200 requests equally distributed for permit and deny. The experiments have been repeated more than 800 times with different policy settings (minimal parameters were 20

Algorithm 1. Insertion algorithm for ACT

```
input : <R1,R2,...,Rn>as P, Resource, current tree as T
Data: handledRules={} set of rules of type <subject,action>
foreach rule R in P do
    if R is of type <subject,action,permit>OR <subject,any decision> then
        if T.subject not exist then
            Add subject in T.subjects
            Copy actions from T.any into T.subject
    if R is of type <subject, action, decision> then
        if decision == permit then
            Add Resource in T.subject.action
            InsCond(R, T.subject.action.Resource)
        Add <subject, action>in handledRules
    else if R is of type <subject, any, decision> then
        actionList = T.decision.subject.actions
        foreach action A in actionList do
            if <subject, A>not in hanledRules then
                if decision == permit then
                    Add Resource in T.subject.A
                    InsCond(R, T.subject.A.Resource)
                Add <subject, A>in handledRules
    else if R is of type <any, action,decision> then
        subjectList = T.decision.subjects
        foreach subject S in subjectList do
            if <S, action>not in handledRules then
                if decision == permit then
                    Add Resource in T.S.actions
                    InsCond(R, T.S.actions.Resource)
                Add <S, Actions>in handledRules
    else if R is of type <any, any, decision> then
        subjectList = T.decision.subjects
        foreach subject S in subjectList do
            actionList = T.decision.S.actions
            foreach action A in actionList do
                if <S, A>not in handledRules then
                    if decision == permit then
                        Add Resource in T.S.A
                        InsCond(R, T.S.A.Resource)
                    Add <S, A>in handledRules
```

Algorithm 2. Request Algorithm for ACT

input : <*subject*, *action*>as *R*, *Resource*, current tree as *PT*
Data: *handledRules*={} set of rules of type <subject,action>
if *R.subject not exist in PT* **then**
  Return set of data objects ids resulting form *PT.any.action* and their conditions
else
  Return set of Data object ids resulting form *PT.subject.action* and their conditions

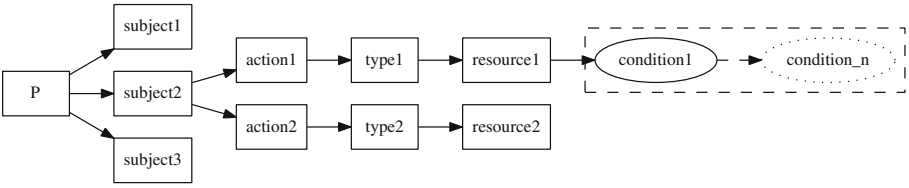

Fig. 2. The extended ACT

users, 20 resources and 800 requests) to confirm the soundness of the results. The policies were generated from the same policy template that contains a rule with a condition requiring for its evaluation an interaction with the PIP to retrieve the value of an attribute. Both ACT and Balana were extended in order to share exactly the same source code for the PIP attribute retrieval operation from a database. The ACT tree structure has been stored in memory.

To simulate a realistic usage in the cloud of the two platforms, we ran our experiments by starting multiple execution threads which request access to resources. For completeness, we also considered the case with a single requestor thread. The number of requests is twice as high as the number of resource/ user combination. This is due to the fact that the requests are intended to cover all the possible cases (i.e., Permit and Deny cases, with and without DB access).

The experimental setting used for the evaluation consisted of an Intel(r) Core(tm) i7 4790 CPU (4 Haswell Cores, 3.75 GHz), 16 GB of System Memory, a 240 GB PCI-EX SSD.

4.1 Performance Tests and Analysis

The first experiment we conducted, depicted in Fig. 3, consisted of measuring the total average time on several executions needed by ACT and Balana to evade 800, 1800, 3200 and 7200 requests when they are issued by single execution thread. The testing scenario of single-threaded sequential query on both of the Balana engine and the ACT shows that the latter always outperforms the former: the difference between ACT and Balana grows from the initial of 5x for 800 requests, to 2 orders of magnitude for 7200 requests. This can be explained by the time

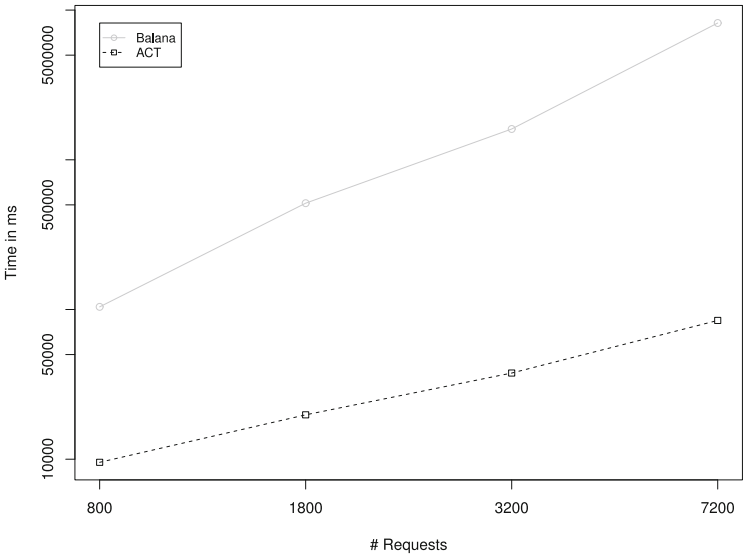

Fig. 3. Comparison of experiments using ACT and Balana

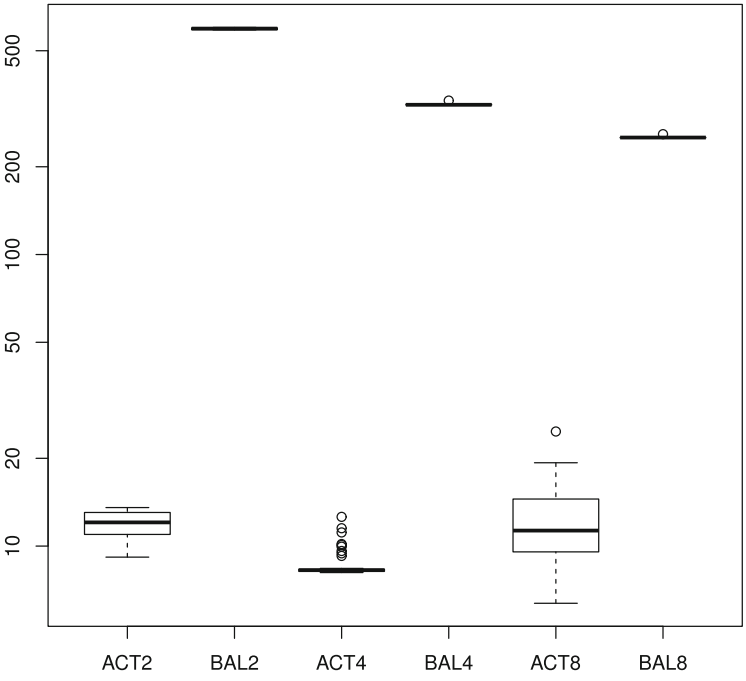

Fig. 4. Execution time in ms for single request using ACT and Balana with 2, 4, and 8 threads.

needed for Balana to scan all the available policies in the repository and trying to match between the request attributes and the policies attributes.

We then conducted a number of experiments in a multi-threaded setting. We present in Fig. 4 the time in ms required for evaluating a single request (out of 7200) using a different number of threads (2, 4 and 8). We calculated these values by collecting the execution times for more than 800 experiments. In all cases, the ACT outperforms Balana, even though the latter scales better in the number of processing threads. On the other hand, the ACT performs slightly more inconsistently in the 8-thread scenario, resulting in slower execution time and a more significant variance as shown by the box plot. Considering that the database used to evaluate the rule condition was deployed on the same execution machine, the fast execution time per single request of ACT as well as the total number of (virtual) cores available on the experimentation server (8), we may conclude that the system resources were almost fully saturated. In the case of Balana, given the higher time necessary for each execution, this effect cannot be observed. The variance of the execution times in Balana experiments is in fact quite limited.

5 Conclusion

Cloud computing needs cloud-designed, performing, efficient and scalable AC solutions. Our proposal extends a cloud AC mechanism based on the XACML standard, the Access Control Tree in order to support instance-based decisions, extending the decision tree with rule conditions. We described our contribution and we evaluated its impact in comparison with an Open Source XACML solution. The results are encouraging and seem aligned with the performance and scalability requirements that we aimed at. As future work we aim at performing a more extensive evaluation and at enhancing further the ACT support for XACML features. Particularly interesting challenges are represented by the evaluation of the impact of rule combining algorithms on the ACT creation process, as well as the possibility to evaluate the impact of multiple conditions (possibily organised in parallel branches) on the performances of the request algorithm.

Acknowledgements. This work was partly supported by EU-funded (FP7/2007–2013) project CoCo Cloud [grant no. 610853].

References

1. Gabillon, A., Munier, M., Bascou, J.-J., Gallon, L., Bruno, E.: An access control model for tree data structures. In: Chan, A.H., Gligor, V.D. (eds.) ISC 2002. LNCS, vol. 2433, p. 117. Springer, Heidelberg (2002)
2. Godik, S., Anderson, A., Parducci, B., Humenn, P., Vajjhala, S.: OASIS eXtensible access control 2 markup language (XACML) 3. Technical report, OASIS (2002)
3. Harnik, D., Kolodner, E.K., Ronen, S., Satran, J., Shulman-Peleg, A., Tal, S.: Secure access mechanism for cloud storage. Scalable Comput. Pract. Exp. **12**(3), 317–336 (2011)

4. Mell. P.M., Grance. T.: SP 800-145. the NIST Definition of Cloud Computing. Technical Report. NIST, Gaithersburg, MD, United States (2011)
5. Popa, L., Yu, M., Ko, S.Y., Ratnasamy, S., Stoica, I.: Cloudpolice: taking access control out of the network. In: Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks, p. 7. ACM (2010)
6. Reeja, S.: Role based access control mechanism in cloud computing using co-operative secondary authorization recycling method. Int. J. Emerg. Technol. Adv. Eng. **2**(10), 25–34 (2012)
7. Shiftehfar, R., Mechitov, K., Agha, G.: Towards a flexible fine-grained access control system for modern cloud applications. In: IEEE CLOUD, pp. 966–967. IEEE (2014)
8. Tang, Z., Wei, J., Sallam, A., Li, K., Li, R.: A new RBAC based access control model for cloud computing. In: Li, R., Cao, J., Bourgeois, J. (eds.) GPC 2012. LNCS, vol. 7296, pp. 279–288. Springer, Heidelberg (2012)
9. Trabelsi, S., Ecuyer, A., Alvarez, P.C.Y., Di Cerbo, F.: Optimizing access control performance for the cloud. In: Proceedings of CLOSER 2014, pp. 551–558 (2014)
10. Younis, Y.A., Kifayat, K., Merabti, M.: An access control model for cloud computing. J. Inf. Secur. Appl. **19**(1), 45–60 (2014)