Flexibility and Scalability in Access Control Systems

Technical University of Denmark

Alex Pellegrini - s132199

***Abstract*—This paper describes two of the most used access control systems i.e. Identity Based Access Control (IBAC) and Role Based Access Control (RBAC). This is going through an analysis with respect to the flexibility and scalability of the mentioned systems.**

I. Introduction

Access control has always been considered one of the most important aspects of security. In computer science this means that system needs to manage interactions between authenticated users and static resurces such as files, directories and devices. This process is also called **authorization**. Access control systems are used to check whether a user is allowed (`authorized`) to access a resources and, if so, what kind of operations the user can perform on it. Usually they are often `policy-based` systems. Once an user has been authenticated and mapped with the right attributes, roles or identity assurance, each time an object is requested, or some actions are requested on it, the policy is evaluated and the permission is then granted or denied. This paper describes how access control policies are enforced in Identity-based (`Discretionary`) and Role-based access control systems.

II. Access Control Policy

Control policies are usally a list of rules that the access control mechanism has to follow in order to know if a request will have success or not. If a request satiesfies all the requirements to access a resource we will say that the permissions, to that request, are **granted**, otherwise they are **denied**. One could see a control policy like a decision tree. As in a machine learning problem we use a decision tree so as to classify a new observation over a set of possible classes, here one could parse a decision tree which only has two classes which are **grant** and **deny**. This has to be done for each request, obviously supposing that such a request has all the attributes in order to be classified by the tree.

Figure 1 shows a trivial example of a possible ”policy tree” used to analyze a request on a resource that can be accessed with `read` and `write` permission by an **Admin** user and only with `read` right for a user of class **A**:

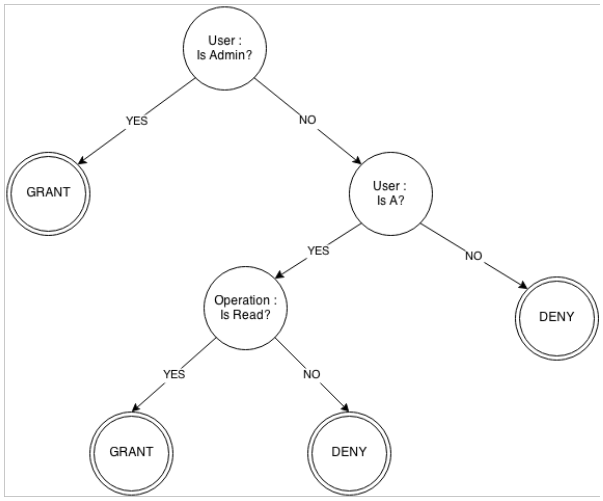

Fig. 1. A tree representation of a control policy

By parsing the tree we can point out that the following request will be granted passing all checks :

Request num.	User Type	Request Type
1	A	`read`

Instead this other request will be denied after the second check (`User : is A?`) :

Request num.	User Type	Request Type
1	B	`read`

III. Identity Based Access Control System

Identity based access control systems (IBAC) are intended to require an owner for each static resource that can decide how to distribute permissons to the other users. IBAC comes with the definition of **subject**, **object** and **access right**. The subject is meant as an active entity that addresses request to access some resources, it could be an user a process or a thread as well. An object is instead a passive entity that could be accessed by an active entity (`subject`) through the

system. An access right is a map between a `subject` and the permissions available on a certain `object`.
Identity-based access control are widely used as security system on operating systems like Unix, Unix-like and Windows NT.

A. Access Control Matrix

Existing relations between `subjects` and `objects` are gathered together in a rectangular matrix where rows are indexed by `subjects` and columns by `objects`. This means that each matrix entry $M[i, j]$ will be the access rights that subject **i** has on object **j**. A request addressed by the subject **s** on the object **o** is granted if and only if the type of the request (e.g. `read,write,execute`) is contained in $M[s, o]$ Adopting a method like this will produce a space usage in the complexity order of $O(nm)$ for a list of subjects with size **n** and an list of objects with size **m** or viceversa.

	Obj1	Obj2	Obj3
Sbj1	`read`	`read,write`	`ex`
Sbj2	`write`		`read,write,ex`
Sbj3	`read`	`read`	`ex`

The access control matrix shown above will produce a space usage of $O(9)$, indipendently if there exist empty entries or not. This matrix would become a sparse matrix very likely and this is a problem from the point of view of performance an maintaning matters and thus is very rarely implemented in real nowadays systems. Instead other two approches are adopted, **Access Control List**s and **Capability List**s.

B. Access Control List

Access control lists (`ACLs`) are a list of permissions attached to an `object` (e.g. a file, directory or device in the case of operating systems). It is usually stored as a table which entries are known as **access control entries** (`ACEs`). Each ACE is a pair containing an user identificator and a list of rights that this user has on the object itself. Figure 2 is an access control list associated to a common object with 5 ACEs.

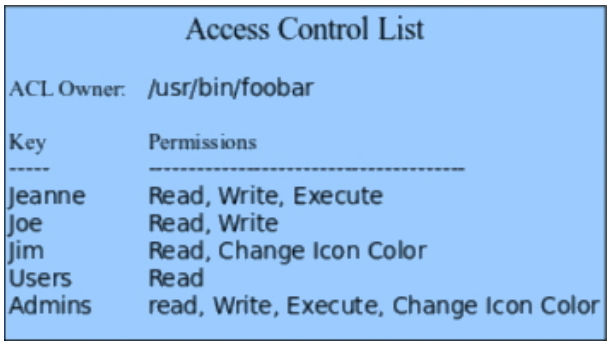

Fig. 2. Rough example of an ACL for a generic `object`

Given an `access control matrix` each column of it could be seen as a single access control list. ACL are a sort of discretionary access control (DAC) which mean that permissions and rights on resources are restricted and these restrictions are based on the requesting `subject` identity. An owner is also requested for each resource and as the name says **discretionary** the owner can decide how to restrict permission on his files. Figure 4 shows an example of how an access control list is displayed on an Unix-like system.

```
-rw-r--r--  1 alex  staff   72 Dec 16 15:21 README.md
-rwxr-xr-x+ 1 alex  staff  226 Dec 16 16:48 setup.sh
 0: group:group deny write,execute
 1: group:admin allow read,write,execute
 2: group:group allow read
drwxr-xr-x  4 alex  staff  136 Dec 16 16:43 src
```

Fig. 3. Example of a unix lke ACL (`setup.sh`)

C. Capability List

Implementation of a capability-based control (i.e. a system using `capability lists`) is not that far from the ACL one. This kind of IBAC is based on the fact that each `subject` is associated with a list of **capabilities**, an entry for each static resource (`object`) present on a computer system. A capability could be see as a key that a subject exhibits when accessing an object and contains constraints which explain in which way the access can be accessed and the righs or operations permitted on it.
A capability list is attached (i.e. associated) to each `subject` and contains an entry for each `object` present in the system. Given an access control matrix $M[i, j]$ each row of it is represents a `capability list`.

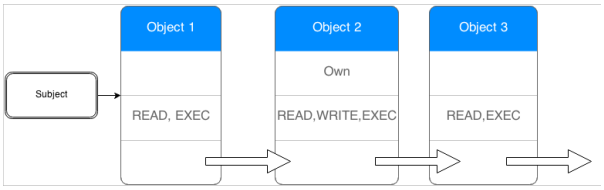

Fig. 4. Example of a unix lke ACL (`setup.sh`)

Lets consider the following python code snippet:

```
1 |    fd = open('foobar.txt', 'rw')
```

this will add a new capability, containing the **file descriptor** of foobar.txt and the set of rights `[read,write]`, to the capability list of the process running the same snippet. In this case the process plays the role of the `subject`.
This kind of approach is quite secure, because the file descriptor is also stored into the kernel memory and thus couldn't be modified by the `subject`.

IV. Role Based Access Control

A newer access control system is being used more widely every day to manage this kind of security matters during the last years. This is the `role based access control` which basically does no longer map authorization, or rather **permissions**, on authenticated user's identity but the role(**s**) covered by the latter.
A **role** can be seen as a duty within an organization for example. Each user can be mapped on ore or more roles based on its duties and, viceversa, a role can have more users. Each role can have one or more permissions. Latter basically are a relation between `access rights` and `objects`. Figure 5 gives a graphic representation of the **RBAC** model.

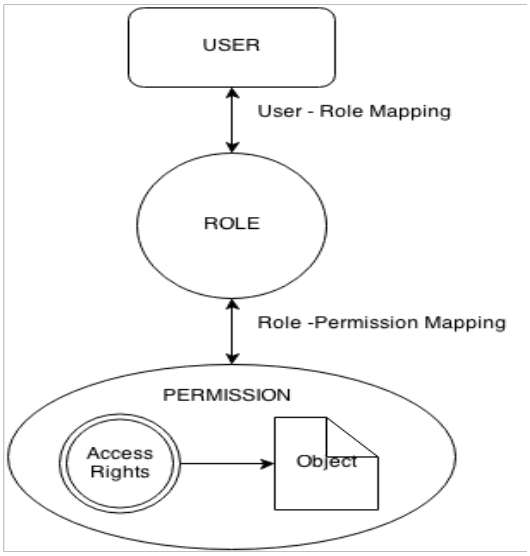

Fig. 5. Illustration of RBAC model

The RBAC model comes with the definition of a set of users U, a set of roles R, a set of permissions P, a mapping between users and roles $UA \subseteq U \times R$ and an assignment relation between roles and permissions $PA \subseteq P \times R$.
There is also the concept of `session` that is a mapping between **u**, **r** and **p** where $u \in U$, $r \in R$ and $p \in P$. The session entity is the reflected `subject` of the Identity-based access control. An user may led more than one session simultaneously each with a subset of the original role set it comes with originally, and therefore a subset of the original permission set. An user **u** request to exercise a permission **p** will be granted, in a session **s**, only if the following holds:

$$Role(\mathbf{s}) \cap Role(\mathbf{p}) \neq \emptyset \tag{1}$$

In an RBAC system a user does not receive its premissions directly but everithing is based on the role it is associated with. In an enterprise with a large number of employees an RBAC system would be very useful as adding a new user will become a matter of associating it with the correct role, and the same holds when updating user's permissions is needed.

It is defined to follow three main rules while working:

1) **Role Assignment** : an user only can perform a transaction if it has been assigned one or more roles.
2) **Role Authorization** : A role must be authorized for the same user or session, it is a verification that the user is actually related to the role itself
3) **Permission Authorization** : A `session` is allowed to perform certain actions only if the permission is authorized for the session's active roles (Equation 1).

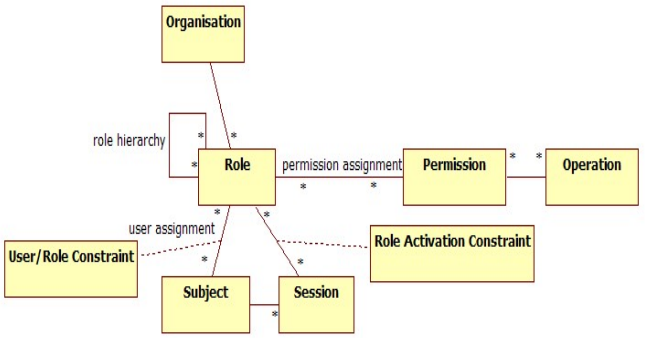

Fig. 6. Representation of an RBAC system (Wikipedia)

Figure 6 shows a possible implementation of an RBAC control system in an organization. As one could see there is also another constraint which is the **Role Activation** constraint. This basically means that during a session two or more conflicting roles cannot be activated together. This is also called **Separation of Duties** (DoS). For example a user can't activate the role that allows him to perform a critical operation and the role to authorize it at the same time.

A. Role Hierarchy

A powerful advantage of this kind of access control system is that a `role hierarchy` could be easly applied. A role hierarchy is some kind of sorting within the set of roles R, or better it is an anti-simmetric and transitive relation

$$RH \subseteq R \times R \tag{2}$$

Lets have a role $r \in R$ this is a senior role, with respect to $r' \in R$ if $Permissions(r') \subseteq Permissions(r)$. r' is thus a junior role with respect to r and this relation is usually written like $r' \leq r$. This means that a `subject` associated with the role r is automatically associated also with the role r'.

In the following example :

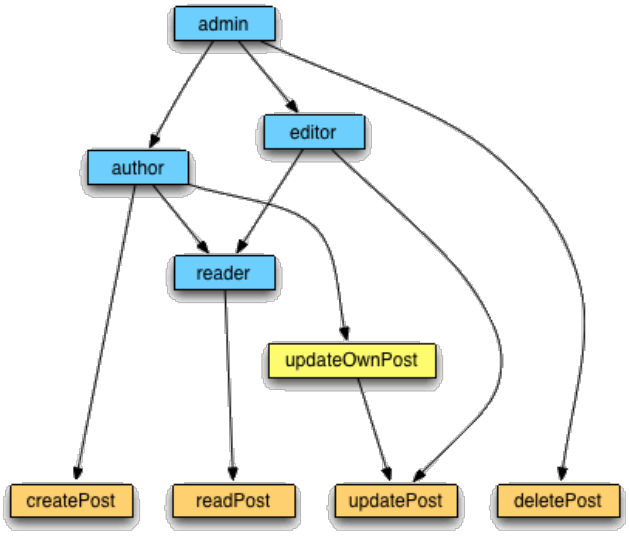

Fig. 7. Organization example

We have a blog system with a role set $admin, author, editor, reader$ and a permissions set $createPosr, readPost, updatePost, deletePost$. In a system like this an user that is only associated with the **reader** role can only read exercise the `readPost`. If a new user, which is supposed to be a post author is added, it is mapped only with the **author** role. Following the hierarchical tree from top to the bottom, starting from the $author$ node one can see that the $reader$ role is automatically associated to the new user without any supplementary operation. An role hierarchical structure like this becomes very helpful to manage new role assignments or adding new subjects to the system because reduces the amount of operations.

V. Flexibility and Scalability Analysis

Suppose we have a big enterprise with a large number of employees and an access control system is needed in order to manage all the matters that concerns working on a computer system. We want to analyze whether it is better to implement an Identity-based Access Control, such as ACLs based or Capability List, or a Role-based Access Control system.
We have seen so far that the biggest difference (at first glance) between the two lies in the fact that in a RBAC system we have an intermediary entity which is the `role`.

$$RBAC = IBAC + ROLES(+RoleHierarchy) \quad (3)$$

The task would be to map employee permissions on the systems `objects` with respect to their job duties and authorities. This could be achieved by implementing an Identity-based Access Control like the ACL one. This means that for each employee belonging to the organization we have to create a list of pairs made by a file descriptor and a set of permissions. This is quite straightforward to do as what we need to do is:

1) for each user, create an ACL and parse all the system `objects`.
2) if the object has to accessible by the user add a pair to the ACL with the Object ID and the permissions.
3) Store the ACL into the system.

In the worst case scenario, where there are n users and m objects, supposing that an ACL entry has constant size and every user can access any file somehow (the object's owner decides this), the implementation will follow a complexity and space time in the order of $O(nm)$. The same holds if we have to add a large set of object into the system. The problem lies in the fact that we have to parse every user's ACL and add a new ACE if the object is accessible by the user. The complexity of such an operation (with a new set of objects with size n) will be $O(n)$, always supposing that creating a new ACE takes $O(1)$. Changing an user's permissions for an object is a matter of look up for the object into the ACL. In the case it is a list structure it will take linear time. When an object is deleted, we have instead to check all the ACLs and remove every ACE that refers to that object. This is done for each user in the system.

Lets say that we want to implement the same system with a Role-based Access Control this time. This means that we have to preapre the set of roles that system needs, and realate them with the correct permissions across the system's objects. Suppose that we still have m objects in our system, and a set of roles with cardinality k where $k \leq n$ and n is the cardinality of the set of users. To map all the roles to the related permissions will take complexity $O(nm)$ again (because of k could grow to n in the worst case).
Moreover a role hierarchy has to be created. This could be seen as a sorted tree like the one shown in Figure 7. To create a tree structure that contains k nodes (roles) and e edges (relationships role-role or role-permissions) takes time $O(k + |E|)$ where —E— is the cardinality of the edges set. We know also that $|E| \leq km$ in the worst case and $k \leq m$, thus:

$$O(k + |E|) = O(k + km) = O(km) = O(nm) \quad (4)$$

Putting all together, the final complexity is:

$$complexity = O(nm) + O(nm) = O(nm) \quad (5)$$

Which is the same as the IBAC implementation (following our assumption). Now, if we want to add a large set of users all what we have to do is to assign one or more roles to each of without the need of parsing the whole set of objects. The same holds when an employee is to be moved or its duties change within the enterprise. We just need to crawl across the 'tree' to retrieve a permission that takes at most $O(h)$, where h is the tree's height (in the case od a binary tree $h = log(n)$).

VI. Conclusion

As a conclusion one could say that an Identity-based Access Control system would be a better choice when the system purpose is to allow a small number of users, for example on an operating system of a personal computer.
For a system that has to hold a large number of users with different authorizations it would be better to adopt a Role-based Access Control system so as to be able to reduce a lot the amount of operations needed.