Towards Session-Aware RBAC Delegation: Function Switch

Meriam Ben Ghorbel-Talbi[1], Frédéric Cuppens[1], Nora Cuppens-Boulahia[1,2], and Stéphane Morucci[2]

[1] Institut TELECOM/Télécom Bretagne
2, rue de la Châtaigneraie, 35576 Cesson-Sévigné, France
[2] SWID, 5 Square du Chêne Germain, 35510 Cesson-Sévigné, France
{meriam.benghorbel,frederic.cuppens,nora.cuppens}@telecom-bretagne.eu
stephane.morucci@swid.fr

Abstract. This paper shows how to extend RBAC sessions with dynamic aspects to deal with user switch. Users can authenticate using their functions which will create a dynamic session and automatically activate a set of privileges associated with this function. A dynamic session can be joined, leaved, restarted and reused by authorized users. Moreover, a user can switch the session to another user in order to continue the task by preserving the working context. We discuss in this paper how to manage users privileges in the dynamic session and how to deal with the switch mechanism.

1 Introduction

The delegation of privileges (permissions, roles) and duties (obligations, responsibilities) has been well studied in recent years and many RBAC extensions have been suggested to deal with these requirements [21,5,14,16]. In this paper we aim to introduce a new concept of dynamic session delegation which we call *switch*. Unlike traditional delegation models we consider that users may be able to "delegate" their whole activated session including their activated privileges and duties, but also their working context which contains running applications and files in use. Hence, the delegatee can continue to run the session exactly like the initial user without impacting the applications or services that are connected to this session, and without any additional authentication burden. This is very useful in many situations such as the continuity of work, where constant online user is required, or in the case of emergency management (e.g., in healthcare or public safety).

We first extend the notion of session suggested in RBAC [15] by dynamic session to enable (1) shareability: more than one user can use the same session (collaborative work), (2) reusability: a session can be reopened while keeping the same states and environments, (3) switchability: a user can transfer his session to another user which can be in a different time space or location. We focus in this paper on this last property and we consider that a session switch involves on itself an authentication, so that the user can directly access to the session

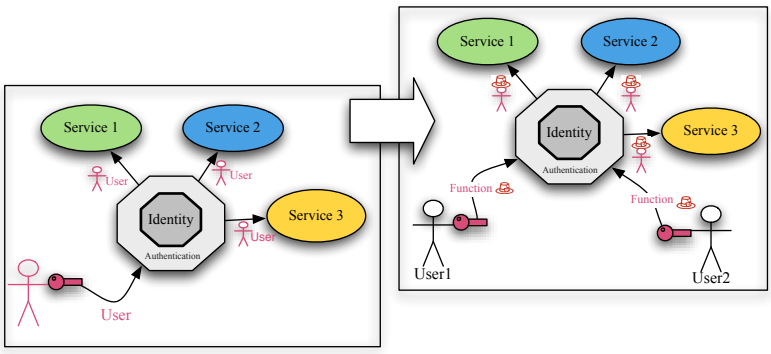

Fig. 1. Function authentication

without additional authentication and can have the whole privileges that are activated in this session. To deal with this aspect we introduce a new concept called *function*. A function can be seen as a job title, such as doctor on duty, that can be used as a virtual identity for the user (see figure 1). This means that, to be authenticated users can provide their function instead of their own identity. Once authenticated, they act as a function and a set of privileges (e.g., roles) that are associated with this function are automatically activated. Hence, if the user switches to another user, this later is authenticated through the same function and can reuse the session with the same environment. Moreover, in the case of collaborative work, many users can be authenticated through the same function and then can share the same session. During the session switch the user activity can be changed, it is what we call *activity switch*. This allows the user to reuse the working context in order to fulfill another task. Moreover, the working environment can be modified, in order to preserve the user privacy, for instance. It is what we call *context switch*.

These new concepts of switch have been introduced in [8] where authors have defined a new model called Smatch (Secure MAnagement of swiTCH) to deal with dynamic session. The Smatch model provides means to specify expressive contextual access control and authentication policies which apply to control functional behavior of dynamic sessions. We base our work on this model and we propose to extend the concept of switch in order to deal with the security administration. In fact, the notion of dynamic session and function authentication have been defined in this model, but the management of the security policy, which encompass users' actions on the dynamic session, has not yet been addressed. We show in this paper how the security administrator can define the security policy related to the function activation, the session joining/leaving, the user switching, the context switching or the function delegation.

This paper is organized as follows. In section 2, we present the basics of the smatch model that we need for our work. In section 3, we give a detailed description of dynamic sessions and how functions are managed in our model. In section 4, we focus on the notion of switch and how to manage the security

policy in this context. Namely, how to manage users privileges in the dynamic session and how to control the switch mechanism (user and context switch). Finally, in section 5, we propose an implementation of our model using Eyeos, an open-source web based Operating System.

2 System Description

Our work is based on the Smatch model [8] which is defined as an extension of the Organization-Based Access Control model OrBAC [6]. This model is based on first order logic with action. First order logic is used to represent the system state at a given time. A system state is represented by a set of ground facts and a set of derivation rules having the form $P1 \wedge ... \wedge Pn \rightarrow P$. The derivation rules are syntactically compatible with Datalog [18]. Negative literals are allowed if it is possible to stratify the derivation rules. Stratifying a Datalog program consists in ordering derivation rules so that if a rule contains a negative literal then the rule that defines this literal is computed first. A stratified Datalog program with negation is computable in polynomial time through the computation of a fix point. Starting from the initial state, the system state can then change due to the execution of actions. Actions are specified through dynamic effect laws having the following form: $A(s, o)$ **causes** P **if** $Q1 \wedge ... \wedge Qn$ where $A(s, o)$ represents the execution of action A by subject s on object o and $P, Q1, ..., Qn$ are negated or unnegated application-dependent predicates.

In this model there are several concepts that are necessary to specify dynamic session and switch. Firstly, the concept of organization is central, which means that several organizations may specify their own security policy. More precisely, we will use the notion of dynamic organization [2] used in the smatch model to deal with dynamic session. Secondly, the administration model proposed by OrBAC [9] is very expressive and provides means to deal with different kinds of delegations [4,3], which is useful to manage the function switch. Moreover, the concept of context is explicitly introduced [6], this means that every security rule (permission, prohibition and obligation) can be associated with a context defined as constraints that a subject must satisfy to activate the rule. This allows us to define dynamic security policy. We give in the following the basics of the Smatch model that we need for our work.

The security policy is specified at the organization level that is independent of the implementation of this policy. Thus, instead of modeling the policy by using the concrete concepts of subject, action and object, it is specified using the roles that subjects, actions or objects play in the organization. The role of a subject is simply called a role, whereas the role of an action is called an activity and the role of an object is called a view. A view is an organizational concept used to structure the policy specification, i.e., a view groups objects to which the same security rules apply. We consider that there are nine basic sets of entities: Org (a set of organizations), F (a set of functions), S (a set of subjects), A (a set of actions), O (a set of objects), R (a set of roles), $\mathcal{A}$ (a set of activities), V

(a set of views) and C (a set of contexts). And we consider the following built-in predicates:

- ***Assign*** is a predicate over domains $Org \times S \times R$. If org is an organization, s a subject and r a role, $Assign(org, s, r)$ means that s can activate role r in org.
- ***Empower*** is a predicate over domains $Org \times S \times R$. If org is an organization, s a subject and r a role, $Empower(org, s, r)$ means that role r is activated by subject s in org.
- ***Use*** is a predicate over domains $Org \times O \times V$. If org is an organization, o is an object and v is a view, then $Use(org, o, v)$ means that org uses o in v.
- ***Consider*** is a predicate over domains $Org \times A \times \mathcal{A}$. If org is an organization, α is an action and a is an activity, then $Consider(org, \alpha, a)$ means that org considers that action α implements activity a.
- ***Hold*** is a predicate over domains $Org \times S \times A \times O \times C$. If org is an organization, s a subject, α an action, o an object and c a context, $Hold(org, s, \alpha, o, c)$ means that within organization org, context c holds between s, α and o.

A security policy corresponds to a set of contextual organization privileges. Abstract permissions are defined using the following predicate:

- ***Permission*** is a predicate over domains $Org \times R \times A \times V \times C$. More precisely, if $auth$ is an organization, r is a role, v is a view, a is an activity and c is a context, then $Permission(auth, r, a, o, c)$ means that $auth$ grants permission to s to perform activity a on view v in context c.

Concrete permissions are derived from abstract permissions when the associated context holds (prohibitions and obligations are similarly defined). Five kinds of contexts have been defined [6]. The Temporal context that depends on the time at which the subject is requesting for an access to the system. The Spatial context that depends on the subject location. The User-declared context that depends on the subject objective (or purpose). The Prerequisite context that depends on characteristics that join the subject, the action and the object. Finally, the Provisional context that depends on previous actions the subject has performed in the system. We can also combine these elementary contexts to define new composed contexts by using conjunction, disjunction and negation operators: &, $\oplus$ and $\bar{}$. This means that if c_1 and c_2 are two contexts, then c_1 & c_2 is a conjunctive context, $c_1 \oplus c_2$ is a disjunctive context and $\bar{c_1}$ is a negative context.

Hierarchies [7] are defined over organizations, roles, activities, views and contexts using predicates $sub\_organization$, sub_role, $sub\_activity$, sub_view and $sub\_context$, respectively. Privileges are inherited through these hierarchies, for instance, permission inheritance is modeled by the following rule:

$permission(org_2, r, a, v, c) \wedge sub\_organization(org_1, org_2)$
$\rightarrow permission(org_1, r, a, v, c)$.

The Smatch model proposes an authentication policy that activates user's privileges according to how users are authenticated: password authentication, strong authentication, one time password or also function authentication. In our work we focus on the function authentication that is defined as follows:

Action $function\_authentication(s, f, pass, org)$
Causes $function\_authenticated(org, s, f)$
If $password(org, f, pass)$

where ***function authentication(s, f, pass, org)*** is an authentication action and $pass$ is the password associated with function f that subject s uses to authenticate in organization org. Obviously, other kinds of function authentication can be required, such as strong authentication using password and token.

3 Dynamic Session

Once authenticated through a function a set of privileges are automatically activated to the user in order to fulfill the task related to this function. And, according to the security policy, the user can switch to another user by preserving or not the activity and the working context. He/she can also allow another user to join the session to fulfill a collaborative task. How to deal with the security policy has not been addressed in the Smatch model, only the states of dynamic sessions and actions that users can perform on these sessions have been described. We give in the following sections our proposition to deal with administration aspects. For this purpose, we extend the function definition proposed in [8] and we give details on how to manage the switch in dynamic sessions.

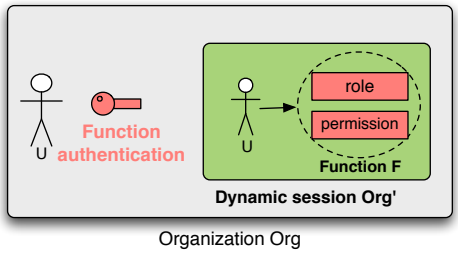

Fig. 2. Dynamic session

We consider that a dynamic session is created when a function is activated by a user, i.e., the user is authenticated through a function. This is different from the "traditional" RBAC session where users are authenticated using their own identity and activate or deactivate their privileges in the session according to their needs. A dynamic session is related to a given task (i.e., a function) so the set of privileges required to fulfill this function is automatically activated in the session. We use in our model the concept of dynamic organization introduced in [2] to define dynamic session. This means that, as described in figure 2, the creation of dynamic session is defined as the creation of a (dynamic)

sub-organization (Org') of the organization in which the function was activated (Org). Privileges associated with the function (F) will be activated for the user (U) in this sub-organization.

As defined in [8] a dynamic session may have different states: active or idle, and users can ask to fulfill different actions in the session according to its state, such as create, join, share, asleep or awake (see figure 3). For the sake of simplicity we only consider in the following the active state and some actions that we need in our work.

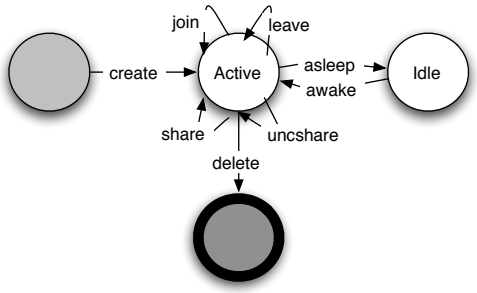

Fig. 3. Session states

We give in the following a description of how functions are defined in our model and the different steps required to create dynamic sessions.

3.1 Function Definition

Each function is associated with a set of roles that will be activated when a session related to this function is created. This is defined using the predicate $Assign$ as follows:

- $Assign(org, f, r)$ means that in organization org role r is assigned to function f.

We also consider that a function is associated with a context which holds when it is activated. We define for this purpose the predicate $function\_context$ as follows:

- $Function\_context(org, f, ctx_F)$ means that in organization org context ctx_F is associated with function f.

This context is used to activate permissions for the user in the dynamic organization, and more precisely permissions to administrate the session. Hence users can restrict the access to some of their data in order to preserve their privacy or to give fewer privileges to other users that join the session. More details about this context are given in the following section.

3.2 Function Activation

We give hereafter the different steps that are needed to activate a function.

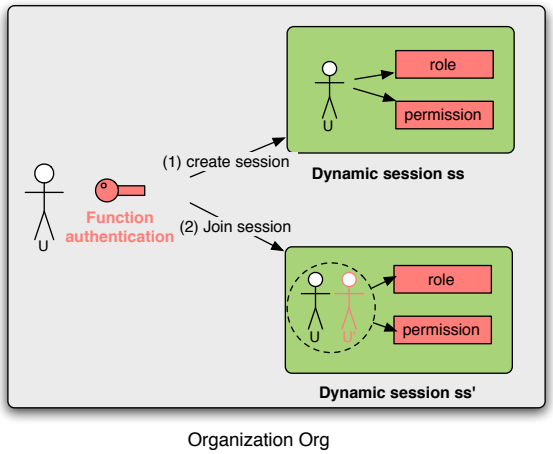

Fig. 4. Dynamic session

Session creation. After the authentication through function f, the user can choose to create a session as follows (see figure 4 part (1)):

Action $create\_session(subj, ss, org)$
Causes $use(org, ss, session) \wedge session\_initiator(ss, subj) \wedge$
$session\_function(ss, f)$
If $function\_authenticated(org, subj, f)$

where $session$ is a special view and $use(org, ss, session)$ means that organization org uses object ss as a session. A session is associated with two attributes: $session\_initiator$: the subject who creates the session and $session\_function$: the function related to this session.

Then the created dynamic session becomes a sub-organization of the parent organization, so that it inherits all the privileges already defined [7]. The inheritance also applies to the assign, use, consider and hold predicates:

$$use(org, ss, session) \rightarrow sub\_organization(ss, org).$$

Session joining. Besides creating a new session, user $subj$ can choose to join an existing session ss' related to function f that is already activated by another user $subj'$ in organization org (see figure 4 part (2)). We consider for this purpose a new attribute $session\_member$ as follows:

Action $join\_session(subj, ss, org)$
Causes $session\_member(ss, subj)$

The session initiator is also considered a session member:

$session\_initiator(ss, subj) \rightarrow session\_member(ss, subj)$.

Obviously, the user can create or join the active session only if he/she is authorized to do so, according to the security policy. The security policy must also specify which users are permitted to authenticate through a given function, and the activation constraints that are related to functions, e.g., exclusive functions or function cardinality. We shall give more details on how to manage the security policy in the following section 4.

Privileges activation. Once the session is created the user will be empowered in roles that are assigned to the activated function:

$use(org, ss, session) \wedge session\_member(ss, subj) \wedge session\_function(ss, f) \wedge assign(org, f, r) \rightarrow empower(ss, subj, r)$.

Note that roles are activated for all the session members. So that when a user joins a session he/she will be automatically empowered on the function roles, and will lose these privileges when he/she leaves the session.

Moreover, the context related to the function is activated within the session. This is defined as follows:

$use(org, ss, session) \wedge session\_function(ss, f) \wedge function\_context(org, f, ctx_F) \rightarrow hold(ss, s, a, o, ctx_F) \wedge session\_member(ss, s) \wedge consider(ss, a, \_) \wedge use(ss, o, \_)$.

This context is used to activate a set of permissions related to the session administration. These permissions can be defined as follows:

$Permission(org, role, activity, view, ctx_F)$.

where *activity* and *view* are administrative activities and views, respectively, and *role* can be defined as a default_role including the session members.

As we said previously, the dynamic session is a sub-organization of the parent organization then it will inherit these permissions. Moreover, the function context holds within the dynamic session so that these permissions will be activated only within the scope of this session. We give in the following more details about how we manage these administrative privileges.

4 Switch management

We present in this section our proposition to manage the security policy. We specify how users can act on functions and dynamic sessions. These actions have been defined in the Smatch model, but how to control these actions has not been addressed. Namely, which users are allowed to authenticate through a function, which users are allowed to join an active session, to delegate their functions or also to switch to another user and under which conditions.

Activating a function. In our model, authenticated users are allowed to activate a given role only if they are assigned to this role:

$$permission(org, authenticated\_user, activate, r, assigned\_role).$$

where context $assigned\_role$ is defined as follows:

$$assign(org, u, r) \rightarrow hold(org, u, activate, r, assigned\_role).$$

Similarly, we consider that users are allowed to activate a dynamic session, i.e., a function, only if they are assigned to this function. This is defined as follows:

$$permission(org, default\_user, activate, f, assigned\_function).$$

where context $assigned\_function$ is defined as follows:

$$assign(org, u, f) \rightarrow hold(org, u, activate, f, assigned\_function).$$

Creating a session. Users are allowed to create a dynamic session only after function authentication:

$$permission(org, authenticated\_user, create, session, activated\_function).$$

where context $activated\_function$ is defined as follows:

$$function\_authenticated(org, subj, f) \wedge session\_initiator(ss, subj) \wedge session\_function(ss, f) \rightarrow hold(org, subj, create, ss, activated\_function).$$

Joining a session. This permission can be specified by the security administrator in order to allow users to join an existing session:

$$permission(org, r, join, session, existing\_session).$$

where context $existing\_session$ is defined as follows:

$$function\_authenticated(org, subj, f) \wedge use(org, ss, session) \wedge session\_function(ss, f) \wedge session\_initiator(ss, subj') \rightarrow hold(org, subj, join, ss, existing\_session).$$

This context means that session ss is already activated by another user $subj'$ and the user $subj$ is already authenticated through function f related to this session.

The security administrator can also specify other conditions using contexts (temporal, spatial, user-declared, prerequisite or provisional contexts [6]). For instance, we can specify that, in the case of an emergency, role *doctor* is allowed to join an existing session related to function *doctor_on_call*:

$$permission(org, doctor, join, session, activated\_session \ \& \ emergency\_doctor).$$

where context *emergency_doctor* holds in the case of an emergency and when the dynamic session is related to function *doctor_on_call*[1]:

$hold(org, \_, \_, \_, emergency) \wedge session\_function(ss, doctor\_on\_call)$
$\rightarrow hold(org, \_, join, ss, emergency\_doctor).$

We can also specify that a user is allowed to join a session only if the initiator is not available, or if a session member is leaving the session.

Other actions on sessions. Similarly to create or to join a session, we can specify permissions to make other actions on sessions, namely to leave, to share, to unshare, to asleep or to awake a session. For instance, in the case of functions where continuity of work is required, a user is allowed to leave a session only if there is other members in this session:

$permission(org, r, leave, session, other\_session\_member).$

where context *other_session_member* is defined as follows:

$session\_member(ss, u) \wedge session\_member(ss, u') \wedge \neg(u' = u)$
$\rightarrow hold(org, u, \_, ss, other\_session\_member).$

User switch. This permission allows a user to assign new permissions to other users in order to join a given session, so that it is called administrative permission. To deal with this kind of permissions we use the administration model proposed by OrBAC [9,4]. This model is based on an object-oriented approach, thus we do not manipulate privileges directly (i.e., Permission and Prohibition), but we use objects having a specific semantic and belonging to specific views, called administrative views. Inserting an object in these views will enable to assign permissions, prohibitions or roles to users. Among these administrative views, we detail in the following the *License view* that is used to specify and manage the security policy. Objects belonging to this view have the following attributes: *Type*: the object type can be a license, a ban or a duty, *Auth*: organization in which the license applies, *Grantee*: subject to which the license is granted, *Privilege*: action permitted by the license, *Target*: object to which the license grants an access and *Context*: specific conditions that must be satisfied to use the license, the ban or the duty. The existence of an object in this view is interpreted as a permission, a prohibition or an obligation according to the object type. For instance, the existence of a valid license is interpreted as a permission by the following rule:

$use(org, l, license) \wedge type(l, license) \wedge auth(l, auth) \wedge grantee(l, r) \wedge$
$privilege(l, act) \wedge target(l, v) \wedge context(l, context)$
$\rightarrow permission(auth, r, act, v, context).$

We consider a sub_view of *license* view called *session_license* as follows:

[1] The prolog symbol _ is interpreted as representing "do not care" condition

$use(org, l, license) \wedge target(l, ss) \wedge use(org, ss, session) \wedge privilege(l, a) \wedge consider(org, a, session\_action) \rightarrow use(org, l, session\_license)$.

where *session_action* is an activity containing actions that users can perform on sessions (create, join, leave, etc.).

This means that the existence of an object in this sub_view is interpreted as the existence of an object in the license view having a session as a target and a session action as a privilege. Users that are allowed to add objects in this sub_view can create new permissions for other users related to sessions, such as a permission to join or to leave a given session.

As previously mentioned, a user switch creates a new permission to join the session, so we consider that a permission to switch in this view is a permission to add a new license with privilege *join*. Permissions to manage switch are defined as follows:

$permission(org, r, switch, session\_license, authorized\_switch)$.

where context *authorized_switch* means that a switch is allowed only for the session members:

$use(org, l, session\_license) \wedge target(l, ss) \wedge session\_member(ss, u)$
$\rightarrow hold(org, u, \_, l, authorized\_switch)$.

We can also add other contexts in order to specify more conditions on the switch. These conditions may concern, for instance, the user who switches the session (according to his/her roles, attributes, etc.), the function related to the session, the user to whom the switch is allowed, e.g., in the case of function *doctor on call* the session can only be switched to a doctor:

$permission(org, r, switch, session\_license, session\_switch \ \& \ doctor\_switch)$.

$auth(l, auth) \wedge target(l, ss) \wedge session\_function(ss, doctor\_on\_call) \wedge grantee(l, u) \wedge assign(auth, u, doctor)$
$\rightarrow hold(org, u, \_, l, session\_license, doctor\_switch)$.

As a result of the user switch a new license is added to the view *session_license* and the user is no longer member of this session (see figure 5):

Action $user\_switch(subj, usr, ss)$
Causes $use(org, l, session\_license) \wedge auth(l, org) \wedge grantee(l, usr) \wedge privilege(l, join) \wedge target(l, ss) \wedge context(l, default\_context) \wedge \neg session\_member(ss, subj))$
If $sub\_organization(ss, org)$

Hence the grantee will be allowed to join the session, using his/her own identity, without the need to perform a function authentication as it is the case usually (see section 3.2). We can also activate an obligation for the grantee to join the session before a given deadline, for example before the other user leaves the session (in the case of continuity of work). Due to space limitation, we do not address the issue of how to deal with obligations in this paper (see [11] for more details about the management of obligations with deadlines).

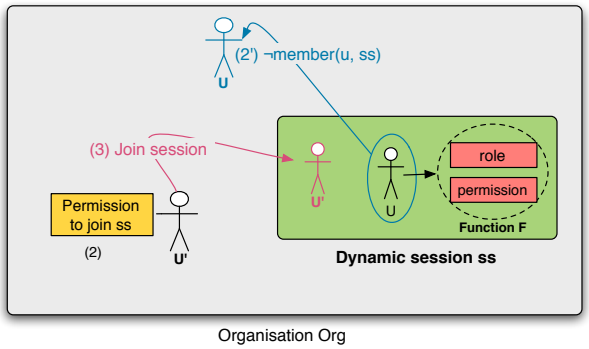

Fig. 5. User switch

The joining user has the same privileges (active roles, permissions, working environment, etc.) as the leaving one. Thus he/she can continue to run the session exactly as before. But, as previously mentioned, the user can modify the session context in order to limit the joining user privileges.

Context switch. To deal with context switch, we consider that users have administrative privileges in the session, namely they are allowed to add prohibitions to other users in order to reduce their privileges. This is managed using the function context introduced previously in section 3.1. This context is used with administrative permissions to specify how users are allowed to administrate dynamic sessions. As defined in section 3.1, context ctx_f holds in the dynamic session after the activation of the function, thus permissions related to it are activated only in the scope of the session. For instance, we can specify that the session initiator of a given function f is allowed to add prohibitions for other users in this session:

$$permission(org, r, add, license\_view, ctx_f \ \& \ c\_prohibition).$$

where ctx_f is a context associated with function f, and $c\_prohibition$ is defined as follows:

$$session\_initiator(ss, u) \wedge type(l, ban) \rightarrow hold(ss, u, \_, l, c\_prohibition).$$

We can also use other contexts to specify more conditions related to the user to whom the session initiator can give prohibitions, and what kind of prohibition he/she can add. For instance, to preserve their privacy, users are allowed to prohibit the access to their personal data. Context switch can also be done automatically, according to the environment change. For instance, we can specify that sensitive information can only be accessed in secured context, e.g., when the user is in his/her office. If the user leaves the office then the access to this information will be automatically prohibited by the access control policy. This can be easily defined in our model thanks to the notion of contextual privileges that is explicitly introduced in the security policy.

Delegating a function Besides switching the session, users can also delegate their functions that are not activated, similarly to role delegation. This means that the grantee will be able to authenticate through the delegated function and create a dynamic session related to this function. For this purpose, we use the same role delegation model proposed in [4]. First, we consider the view $Function\_assignment$ as follows:

$use(auth, fa, function\_assignment) \wedge auth(fa, org) \wedge assignee(fa, r) \wedge assignment(fa, f) \rightarrow assign(org, f, r)$.

This means that the existence of a valid object fa in this view is interpreted as an assignment of function f to role (or subject) r in organization org. Then, to deal with function delegation, we consider a sub-view of $function\_ assignment$ called $function\_delegation$. Objects belonging to this view inherit the semantic and the attributes of view $function\_assignment$, but, also have an additional attribute called *Grantor*: the subject who is delegating the function. Thus inserting an object in this view will enable an authorized grantor to delegate a function to a grantee. The security administrator can specify which users/roles are allowed to delegate their functions and in which contexts as follows:

$permission(org, r, delegate, function\_delegation, context)$.

The security administrator can also specify how much the function can be delegated and/or re-delegated (multiple and multi-step delegation, respectively), if the delegation is temporary or permanent, and if the grantor keeps the function or not after the delegation (transfer). More details about delegation are given in [4]. Hence to summarize this section, the security policy controls all the users actions on the session such as the activation, the joining, the switch and the delegation of functions. Authorized actions must also satisfy the global constraints specified by the security administrator, i.e., the separation of duty policies related to the activation/deactivation of functions, the joining/leaving of sessions, etc. For instance, we have to deal with dynamic separation of duties in the case of the user switch. Namely, when a user joins a session s_1, his/her roles that are activated in another session s_2 can conflict with roles activated in s_1. In this case, we can choose to refuse the switch or also to activate a pre-obligation to the user in order to leave the session s_2 before allowing him/her to join session s_1 (how to manage pre-obligations is presented in [10]). Many other constraints can be defined to deal with functions similarly to roles like cardinality, exclusive functions, etc. This is an important issue to consider and we aim to address it in our future work.

5 Switch enforcement

For the implementation of our model we choose to use Eyeos [12], an efficient open-source web-based Operating System, following the cloud computing concept. It enables collaboration and communication among users. We have modified this OS to include a dedicated access control mechanism that overrides the

Eyeos access control function to provide enhanced capabilities and demonstrate the switch concept. Access control policies are expressed using MotOrBAC [1] a tool developed at Telecom Bretagne that implements the OrBAC model. It provides a user-friendly interface to specify and manage the security policy.

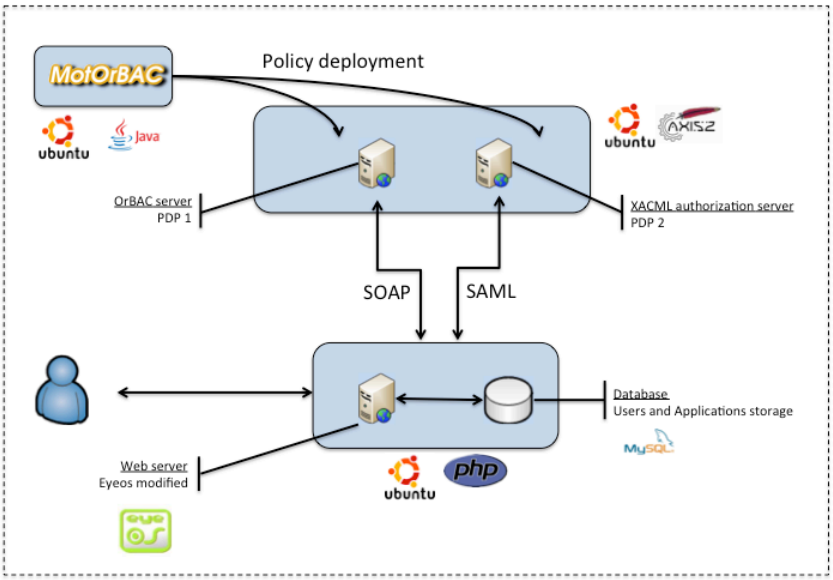

Fig. 6. Architecture

As specified in figure 6, these policies are enforced using a specific Policy Enforcement Point (PEP) and two Policy Decision Points (PDP), only one PDP is active at a time. A PEP has been implemented in Eyeos as a replacement of current access control module. Two OrBAC-based PDPs, with two kinds of requesting methods: 1) A PDP which is queried using a standardized XACML protocol. OrBAC policies are translated into XACML and processed by the Swid XACML server. This PDP has the advantage of handling standard protocols. By contrast, XACML policies cannot express fine-grained access-control policies with complex context processing. 2) A PDP which is queried using dedicated Web-services. OrBAC policies are directly interpreted for each query. This PDP leverages the power of the OrBAC model, making it possible to have advanced security policies (through expressive context evaluation).

Swid software agents are integrated into Eyeos to handle communication (either SAML or Web-service) between Eyeos access control mechanism and PDPs. Using this application, users can be authenticated through a function, and get an access to a limited set of applications, depending on their rights (privileges that are assigned to the activated function). Several users can share the same session once authenticated through the same function. Users can close their sessions or perform a user switch. This second behavior is similar to the first one, but in addition it allows another user to join the session. In these two cases,

the session can be resumed by authorized users, and all applications previously launched are displayed. Moreover, all data written by the previous user are kept, making it possible to append some additional information. The session context can change after the user switch, for instance, we may consider that if the user location is not in a secured area, then the security policy forbids the access to confidential information and applications. Note that the context can also be modified without the user switch, for instance, when the user moves from his/her office to another location, some applications will be prohibited or closed. Or also when an emergency occurs, the user will have an access to additional privileges in order to deal with urgent situations. As future work, we aim to extend this application with more administration features that we have discussed in this paper. For instance, in our model users are able to perform a context switch by forbidding the access to their personal data to the other session members. This will update the security policy by adding prohibition rules.

6 Discussion and conclusion

To the best of our knowledge, we have addressed in this paper new issues that have not been previously considered by access control models. Our concept of dynamic session is different from traditional RBAC sessions [15] since, it supports new features namely, shareability, reusability and switchability. In our model, a user may create a dynamic session and starts the execution of a task, suspend the session, reopen it, and continue the execution of this task in another context. Also, using the switch operation, this task may be continued by another user. Moreover, we have introduced the concept of function that involves both authentication and access control mechanisms to ease the switch operation. When function authentication is used a set of privileges that are needed to perform the related task, are automatically activated for the user. This is different from the concept of task defined in task based RBAC models such as [20,17,19] since, in these models, the dynamic behavior of our approach is not addressed.

In [13] authors propose the concept of capability delegation, where a capability represents a self-authenticating permission to access a specified object in permitted operations. Our concept of function switch is different form delegation, since the working context of the initial session will not be lost when the joining user activates the dynamic session, which is not the case in "classical" delegation. During the switch, the context can change according to the security policy, to preserve user privacy or for security reasons. The user activity can also change to fulfill another task by preserving the working environment. Another switch issue that has not been addressed in this paper is the organization switch, when the user hands over to another user belonging to another organization. This is the most complex case to manage since the user, who is targeted by the switch, will not be probably assigned to the "same" role as the original user. We aim to further investigate this feature in future work. We also aim to study separation of duty policies related to the activation and deactivation of dynamic sessions especially in the case of user switch.

Acknowledgments. This research has been supported by the RoleID project. Role-ID is a european research project funded by Eureka, ITEA 2 programe. The project is coordinated by EADS Defense and Security Systems, France.

References

1. Autrel, F., Cuppens, F., Cuppens, N., Coma, C.: MotOrBAC 2: A Security Policy Tool. In: SARSSI (2008)
2. Autrel, F., Cuppens-Boulahia, N., Cuppens, F.: Reaction Policy Model Based on Dynamic Organizations and Threat Context. In: DBSec (2009)
3. Ben-Ghorbel-Talbi, M., Cuppens, F., Cuppens-Boulahia, N., Bouhoula, A.: An Extended Role-Based Access Control Model for Delegating Obligations. In: TrustBus (2009)
4. Ben-Ghorbel-Talbi, M., Cuppens, F., Cuppens-Boulahia, N., Bouhoula, A.: A Delegation Model for Extended RBAC. The International Journal of Information Security (IJIS) 9(3) (June 2010)
5. Crampton, J., Khambhammettu, H.: Delegation in Role-Based Access Control. International Journal of Information Security (September 2008)
6. Cuppens, F., Cuppens-Boulahia, N.: Modeling Contextual Security Policies. International Journal of Information Security 7(4) (2008)
7. Cuppens, F., Cuppens-Boulahia, N., Miège, A.: Inheritance hierarchies in the OrBAC Model and application in a network environment. In: FCS (2004)
8. Cuppens, F., Cuppens-Boulahia, N., Nuadi, M.: Smatch Model: Extending RBAC Sessions in Virtualization Environment. In: ARES (2011)
9. Cuppens, F.C., Cuppens-Boulahia, N., Coma, C.: Multi-Granular Licences to Decentralize Security Administration. In: SSS/WRAS (2007)
10. El-Rakaiby, Y., Cuppens, F., Cuppens-Boulahia, N.: From Contextual Permission to Dynamic Pre-Obligation. In: ARES (2010)
11. Elrakaiby, Y., Cuppens, F., Cuppens-Boulahia, N.: Formal enforcement and management of obligation policies. Data & Knowledge Engineering (2011)
12. EYEOS: http://www.eyeos.org/
13. Hasebe, K., Mabuchi, M., Matsushita, A.: Capability-Based Delegation Model in RBAC. In: SACMAT (2010)
14. Ray, I., Toahchoodee, M.: A Spatio-temporal Access Control Model Supporting Delegation for Pervasive Computing Applications. In: TrustBus (2008)
15. Sandhu, R., Coyne, E.J., Feinstein, H.L., Youman, C.E.: Role-Based Access Control Models. IEEE Computer 29(2), 38–47 (1996)
16. Schaad, A., Moffett, J.D.: Delegation of Obligations. In: POLICY (2002)
17. S.Oh, S.Park: Task-Role-based Access Control Model. Information Systems 28 (2003)
18. Ullman, J.D.: Principles of Database and Knowledge-Base Systems: Volume II: The New Technologies. W. H. Freeman & Co., New York, NY, USA (1990)
19. Yao, L., Kong, X., Xu, Z.: A Task-Role Based Access Control Model With Multi-Constraints. In: NCM (2008)
20. Zhang, L., Luo, L., Zhang, L., Geng, T., Yue, Z.: Task-Role-Based Access Control in Application on MIS. In: APSCC (2006)
21. Zhang, X., Oh, S., Sandhu, R.: Pbdm: A Flexible Delegation Model in RBAC. In: SACMAT (2003)