Journal of Universal Computer Science, vol. 15, no. 3 (2009), 538-554
submitted: 3/1/09, accepted: 28/1/09, appeared: 1/2/09 © J.UCS

DS RBAC – Dynamic Sessions in Role Based Access Control

Jörg R. Mühlbacher
(Johannes Kepler University Linz, Austria
muehlbacher@fim.uni-linz.ac.at)

Christian Praher
(Johannes Kepler University Linz, Austria
praher@fim.uni-linz.ac.at)

Abstract: Besides the well established access control models, Discretionary Access Control (DAC) and Mandatory Access Control (MAC), the policy neutral Role Based Access Control (RBAC) is gaining increasingly momentum. An important step towards a wide acceptance of RBAC has been achieved by the standardization of RBAC through the American National Standards Institute (ANSI) in 2004.

While the concept of sessions specified in the ANSI RBAC standard allows for differentiated role selections according to tasks that have to be performed by users, it is very likely that more roles will be activated in a session than are effectively needed to perform the intended activity. Dynamic Sessions in RBAC (DS RBAC) is an extension to the existing RBAC ANSI standard that dynamically deactivates roles in a session if they are not exercised for a certain period of time. This allows for the selection of an outer-shell of possibly needed permissions at the initation of a session through a user, while adhering to the principle of least privilege by automatically reducing the effective permission space to those roles really exercised in the session.

Analogous to the working set model known from virtual memory, only the minimal roles containing permissions recently exercised by the user are left in a session in the DS RBAC model. If the user tries to access a role that has aged out due to inactivity, a role fault occurs. A role fault can be resolved by the role fault handler that is responsible for re-activating the expired role. As will be presented in this paper, role re-activation may be subject to constraints that have to be fulfilled by the user in order to re-access the aged role.

Key Words: security, Role Based Access Control, ANSI RBAC, session, least privilege

Category: D.4.6, K.6.5, L.4.0

1 Introduction

The relatively young model of Role Based Access Control (RBAC) has successfully proven to simulate more traditional access control models, like e.g. Discretionary Access Control (DAC) and Mandatory Access Control (MAC) [Osborn et al. 2000] and even improve on them in certain respects. Most importantly, RBAC eases the administration of a security system, by combining individual users into roles and managing permission assignments on a role basis rather than on a per user basis. This offers the benefit of dealing with less entities and reflecting the organizational structures more natural[Barkley 1997].

Since its inception in the early 1990ies, RBAC has been a major field of both academic and corporate research and many different flavors of RBAC tailored for specific fields of application exist. In 2004, the American National Standards Institute (ANSI) consolidated the most important characteristic of RBAC and released them under the ANSI INCITS 359-2004 standard[ANSI RBAC 2004]. Amongst the major components of ANSI RBAC are a mandatory session concept, as well as optional hierarchies and separation of duties.

From the set of assigned roles, a user can choose or is automatically provided with a subset of roles that are activated in a session for conducting one or more specific tasks. Ideally, a session would contain only the permissions really needed to fulfill exactly the tasks it was created for, which limits the potential dangers of deliberate or accidental abuse of the system. In practice though, users will rarely voluntarily stick to only the smallest possible set of permissions needed to perform a certain task. In contrast, it is highly likely that users will activate as much roles as possible to be able to perform as much operations at once without having to re-authenticate. Such a behavior can be observed e.g. on simple stand-alone PC's, where users log in as *Administrator* or *root*, despite of only exercising normal user permissions. This can seriously violate the principle of least privilege, which requires that *"a user be given no more privilege than necessary to perform a job"*[Ferraiolo and Kuhn 1992].

Dynamic Sessions in Role Based Access Control (DS RBAC) tries to provide a solution to this problem, by allowing a session to dynamically shrink and expand according to the user's actual resource usage behavior. A user is allowed to activate as many roles at the start of the session, as she/he thinks, but these roles may expire in the course of the session if they are not continuously exercised. The aging timeout of a role can be set by the system administrator according to management decisions and general organizational policies. After some time, a session containing only the *minimum roles* holding permissions recently exercised by the user, should be left. As the term minimum role suggests, DS RBAC mandates a well ordering on roles, which allows the system to automatically choose the smallest one from a set of roles.

What is important is that the model of Dynamic Sessions in RBAC proposed in this paper tightly integrates and builds upon the standard ANSI RBAC model[ANSI RBAC 2004]. As will be shown, the concept could be realized with only relatively few changes to the standard on RBAC defined by the ANSI.

2 Principles of DS RBAC

In some respects, the proposed model of DS RBAC can be seen as an inverse analogy to the working set model for virtual memory, introduced by Peter Denning [Denning 1968]. The working set model states that a process only runs efficiently if its working set is held in RAM. The working set of a process is defined to be the set of its most recently used pages, where most recently refers to a working set window, a specified time interval. It is statistically very likely that a running process will access these most recently used pages also in the near future. If an addressed page currently is not member of the working set and thus not held in RAM, then a page fault is raised, which activates a page fault handler. Its main task, among others, is to fetch the missing page to memory.

Dynamic Sessions in RBAC tries to transfer this idea to RBAC. The working set in DS RBAC is a collection of roles, which the user most recently exercised. More precisely, the working set in DS RBAC is the set of roles, which consists of the *minimum roles* holding permissions the user currently exercised.

If the user tries to access a permission that is not currently contained in one of the roles in the working set, a role fault occurs. It triggers the role fault handler that may re-activate the *minimum role* containing the needed permission. As will be described later, the role fault handler either may impose various constraints on the re-activation of the role, or simply activate it without user intervention.

2.1 Extensions of the Role Concept

DS RBAC extends the role concept known from ANSI RBAC, to a tuple, consisting of a unique role identifier and a time to live (*ttl*) value: $(role\_id, ttl)$

The identifier *role_id* is just a name for the role, needed to distinguish it from other roles systemwide. This is identical to the definition of a role under the ANSI RBAC standard, which treats a role (a part from its relations to users and permissions) as just being an abstract name, simply a word over an alphabet.

The time to live specifies a validity time for each role. The following sections describe the concept of time to live and role aging in more detail.

2.2 Well ordering of roles

There is an underlying well ordering on the set of roles $\{r_i\}$, defined by the ordinary "less than" relation "$<$". This ordering allows to create an hierarchy of roles in terms of their mightiness. If the user wants to exercise a certain permission that is contained in two or more roles, the system takes the least powerful one. This role is not necessarily the one with the fewest permissions, but rather the one which would cause the least damage if it was used adverseley by an attacker, or accidentially by an user.

The order "$<$" can be established by the administrator following a particular privilege policy or computed automatically e.g. by summing up weighted operations $g(op_i)$ and objects $w(ob_i)$ according to a certain ranking g for operations op_i and level w for objects ob_i. When subsets of roles turn out to receive the same weight, they may be arranged arbitrarily (e.g. lexicographically), so that $r_i < r_j$ or $r_j < r_i$ always holds. The ranking g of operations op_i, as well as the level w of objects ob_i could be defined by the administrator or default to system specific values.

Assume role r refers to objects ob_1, ob_2 with the permissions $\{(read, ob_1), (read, ob_2), (write, ob_1)\}$ and there are weights $g(ob_1)$ and $g(ob_2)$ of objects and weights $w(read), w(write)$ for the operations $\{read, write\}$ respectively, then the rank h of r is given by:

$$h(r) = w(read) \otimes g(ob_1) + w(read) \otimes g(ob_2) + w(write) \otimes g(ob_1)$$

where $\otimes$ could be the ordinary multiplication operator.

A possible administrative facilitation could be to classify the relatively large set of objects by $\{critical, uncritical\} = \{c, u\}$ so that $w(ob)$ yields either c or u, with $c >> u$. The compared to objects relatively low number of operations could be each assigned an individual value by the ranking function g.

It is important to emphasize that the well ordering of roles concerning their power should be independent of RBAC hierarchies and apply to all roles in the system. For a clear discrimination of these two relations, the symbol "$\succeq$" is used for RBAC hierarchies and "$<$" is used for the role well ordering.

2.3 Role Aging

The main concept of DS RBAC which makes sessions dynamic is *role aging*. The aging timeout of a role $r = (role\_id, ttl)$ is specified by the time to live value ttl: $r.ttl$. If not specified by the administrator, the default ttl is ∞. This means that a role r with $r.ttl = \infty$ is not subject to aging.

It is important to differentiate the global ttl given to a role in the system and the concrete activation timestamp ts a role has within a user's session. The ttl associated to the role defines the maximum period of inactivity any role can have, without being subject to expiry. Within the user's session, the activation timestamp for every role is stored in the *session_roles* mapping. A role's activation timestamp is the current system time at the time of activation of the role in a concrete user session, or the time when one of the permissions held by the role was last exercised in that session. A concrete instance of a role in a user's session is considered expired if its activation timestamp plus its administratively associated time to live (ttl) is smaller than the current system time.

There is one special role $rdefault$, whose purpose will be explained later, which never expires and belongs to every user session.

From an implementation perspective, expiry of a role could be checked eagerly by means of a daemon service, or lazily at access time in the $CheckAccess$ function. Eagerly checking of expired roles could be done in a fashion similar to Garbage Collection (GC) in memory managed programming languages.

2.4 Role Fault

A *role fault* occurs, if a user tries to access a permission implied only by roles that are already expired in the current session. It is important to emphasize that a role fault only occurs if at least one role holding the needed permission is part of the user's current session. If the user never activated such a role, the behavior is the same as in the standard RBAC case, which means that access is denied. In case of a role fault, a role fault handler is triggered, which may require some additional qualification from the user in order to pass the access check and to re-activate the role.

2.5 Role Fault Handler

The *role fault handler* is called by the $CheckAccess$ function whenever a user tries to exercise a permission held only by expired session roles. It is thus the principal mechanism for resolving possible permission conflicts that arise because of expired roles. Not all roles are necessarily subject to re-activation constraints, but typical constraints might be: Re-authentication of the user (e.g. re-type password, slide over the fingerprint scan, query for security token given at start of session, ...), or log-file activation.

If the role fault handler constraint can be answered by the user, the role fault handler signals back to the $CheckAccess$-function that the user should be allowed to perform the desired operation on the object.

If the associated constraint can not be fulfilled by the user, the role fault handler returns to the $CheckAccess$-function that access should not be granted.

3 Formal Description of the DS RBAC Core Model

The following is a formal specification of the DS RBAC Core model, which is a direct extension of the ANSI RBAC Core model[ANSI RBAC 2004]. An ample discussion about the side effects of DS RBAC on the other three ANSI RBAC components is provided in the next section "Side Effects on other RBAC Components".

- $USERS$, OPS, and OBS refer to users, operations and objects respectively.
- $T \subset \mathbb{N}_0$, a set of possible timeticks represented as natural numbers, including 0.
- Let be α an alphabet, then $role\_id \in \alpha^*$ is called a role name, e.g. $role\_id$ is a word over α. For convenience we use $role\_id_i, i = 1, 2, \ldots$ for such names only, unless we name a specific role explicitly.
- $NAMES = \{role\_id_i | i = 1, 2, \ldots n \wedge role\_id_i \in \alpha^*\}$
  Therefore $role\_id_i \neq role\_id_j$ for $i \neq j$, because $NAMES$ is a set.
- $ROLES = \{r_i | i = 1, 2, \ldots n\} = \{(role\_id, ttl) | role\_id \in NAMES \wedge \ ttl \ \in T\}$, the tuple contains: A system wide unique role name $role\_id$ and a time-to-live (ttl) given in timeticks t specifing the expiry time of the role.
- $assigned\_ident(r : ROLES) \rightarrow NAMES$, a function that returns the identifier of the given role r.
- $assigned\_ttl(r : ROLES) \rightarrow T$, a function that returns the ttl defined for a given role r.
- $GRO \subseteq ROLES \times ROLES$, General Role Order (GRO) is a well ordering defined on all roles r_i, r_j, such that $r_i < r_j$ or $r_j < r_i$ for $i \neq j$ always holds. This ordering allows to unambiguously relate roles to each other in terms of their mightiness. It could e.g. be computed automatically as described earlier.
- $current\_system\_time \in T$, the current system time.
- $UA \subseteq USERS \times ROLES$, a many-to-many mapping user-to-role assignment relation.
- $assigned\_users(r : ROLES) \rightarrow 2^{USERS}$, the mapping of role r onto a set of users.
  Formally: $assigned\_users(r) = \{u \in USERS | (u, r) \in UA\}$
- $PRMS = 2^{(OPS \times OBS)}$, the set of permissions $p = (op, ob)$.
- $PA \subseteq PRMS \times ROLES$, a many-to-many mapping permission-to-role assingment relation.
- $assigned\_permissions(r : ROLES) \rightarrow 2^{PRMS}$, the mapping of role r into a set of permissions.
  Formally: $assigned\_permissions(r) = \{p \in PRMS | (p, r) \in PA\}$
- $Op(p : PRMS) \rightarrow 2^{OPS}$, the permission to operation mapping, which gives the set of operations associated with permission p.

- $Ob(p : PRMS) \rightarrow 2^{OBS}$, the permission to object mapping, which gives the set of objects associated with permission p.
- $SESSIONS =$ the set of all sessions in the system.
- $session\_user(s : SESSIONS) \rightarrow USERS$, the mapping of session s onto the corresponding user.
- $user\_sessions(u : USERS) \rightarrow SESSIONS$, the mapping of user u onto the corresponding sessions.
- There is one and only one special role $rdefault \in ROLES$ with $rdefault = (rdefault\_id, \infty)$. $\forall u \in USERS \rightarrow (u, rdefault) \in UA$. The role $rdefault$ is implicitly assigned to each user and because of $assigned\_ttl(rdefault) = \infty$ it will never expire.
  In addition we demand the following: the set of permissions assigned to $rdefault$ should be as small as possible, depending on the system where the roles are defined. Typically $rdefault$ would contain permissions to login and logout and to activate expired roles.
- $session\_roles(s : SESSIONS) \rightarrow 2^{ROLES \times T}$, the mapping of session s to a set of roles. For every role r in the session s, the timestamp of activation is stored. We note that with respect to the RBAC ANSI-Standard there is an extension: Instead of $session\_roles(s : SESSIONS) \rightarrow 2^{ROLES}$ we use $session\_roles(s : SESSIONS) \rightarrow 2^{ROLES \times T}$ in order to provide information when a role (instance) has been activated or used most recently.
  Implicitly, every session contains the minimum default role $rdefault$ that never expires.
  Formally:
  $session\_roles(s_i) \subseteq \{(r, ts) | r \in ROLES \wedge ts \in T \wedge (session\_user(s_i), r) \in UA\}$
- $avail\_session\_perms(s : SESSIONS) \rightarrow 2^{PRMS}$, the 'outer shell' of permissions available to a user in a session. We note that due to aging, not all of these permissions must always be available to the user throughout the session. Formally:
  $avail\_session\_perms(s) = \bigcup_{(r,ts) \in session\_roles(s)} assigned\_permissions(r)$
- $effective\_session\_perms(s : SESSIONS) \rightarrow 2^{PRMS}$, the effective permissions available to a user in a session. This relation includes only those permissions that are implied by roles that have not expired. Formally:
  $effective\_session\_perms(s) =$
  $\bigcup_{(r,ts) \in session\_roles(s) \wedge (ts + assigned\_ttl(r)) \geq current\_system\_time}$
  $assigned\_permissions(r)$

3.1 CheckAccess

CheckAccess is the main method that decides whether access to a resource is given to a user or not. Within the DS RBAC model it is also responsible for identifying role faults and triggering the role fault handler for properly resolving the role fault. The role fault handler in turn signals back to the *CheckAccess* function, whether an attempt to re-activate a role for a user was successful or not. The basic procedure of *CheckAccess* is as follows (see flowchart in figure 1):

- Access is denied, if the needed permission $p = (op, ob)$ is not contained in any of the roles in the user's current active session. Like in the ANSI RBAC standard, the active sessions of an user are defined by the *sesssion_roles* function.

- If the needed permission is provided by one of the roles in the users's current active session, the following two possibilities exist:
  - If there is at least one *active* role holding the needed permission, access is granted. An active role is a role that has not expired in a session. Amongst *all* roles in the current user session, the minimum role that contains the needed permission is identified. The minimum role is different from $rdefault$. The timestamp of the minimum role becomes updated with the current system time. It does not matter if the role that initially allowed the access because of still being active is different from the role that eventually gets its timestamp updated. Also, it does not matter if the minimum role is expired. Because of the minimum characteristic of the chosen role, it is always the role with the least privileges from all roles in the current user's session that gets updated and implicitly re-activated if necessary.
  - If there is no active role holding the needed permission, a role fault occurs. The role fault triggers the role fault handler that is responsible for re-activating the expired role. The role fault handler can implement various methods of how an aged role can be re-activated. Depending on the outcome of the role fault handler, access is either granted or denied.
    - Access is granted, if the role fault handler can be answered positively. In this case, it returns true to the *CheckAccess*-function, which in turn gives access to the resource. Like in the case with the active role, the minimum role amongst all roles of the current user session (*sesssion_roles*) is identified. The timestamp of this role gets updated and all permissions associated to it are again available to the user. As mentioned, a role fault handler could also implicitly provide

a positive return value (E.g. in case of a log-file activation, the user would not even notice the aging of the role and re-activation through the role fault handler).

∗ Access is denied, if the role fault handler can not be answered. In this case the role fault handler returns false to *CheckAccess*.

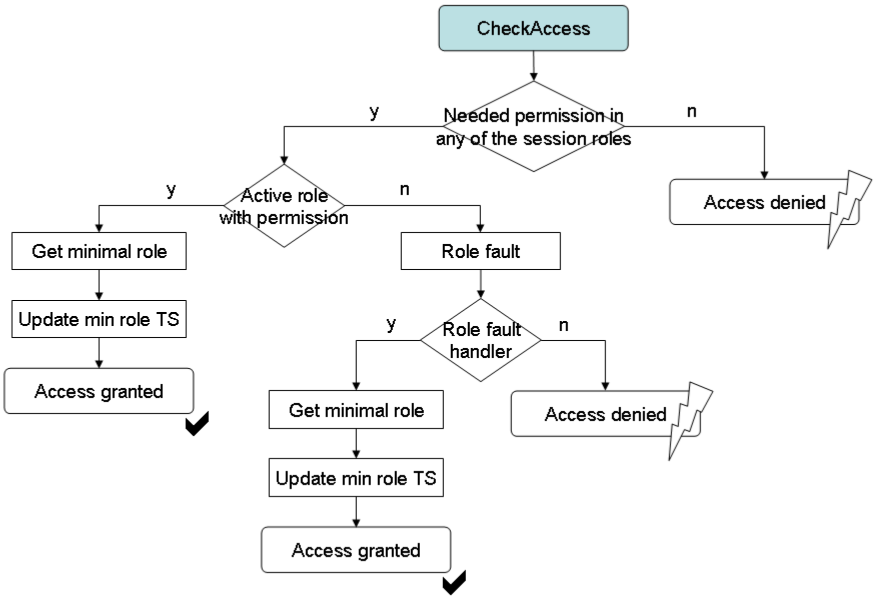

Figure 1: Flow chart of check access

```
boolean CheckAccess (session, operation, object) {

  p = (operation,object)
  p ∈ PRMS

  /*
    A list of all active roles in the current session.
  */
  active_session_roles;

  /*
    A list of all expired roles in the current session.
  */
  expired_session_roles;

  /*
    Identify the minimum access role.
    This role is, according to the well ordering, the smallest role
    in the set of session roles that contains the needed permission.
  */
  access_role = GetMinAccessRole(session_roles(session), perm);

  /* There is an active role with the needed permission. */
  if (∃(r, ts) ∈ active_session_roles : (p, r) ∈ in PA) {
    access_role.ts = current_system_time;
    return true;
  }

  /*
    Role fault case.
    There is no active role containing the needed permission, but only
    an expired one.
  */
  if (∃(r, ts) ∈ expired_session_roles : (p, r) ∈ in PA) {
    if (RoleFaultHandler(access_role)) {
        access_role.ts = current_system_time;
        return true;
    }
    /* Acces denied - Role fault handler could not be answered */
    return false;
  }

  /* Access denied - Permission not at all in user session. */
  return false;
}
```

Listing 1: CheckAccess in pseudo-code

4 Side Effects on other RBAC Components

As will be presented in detail in this section, DS RBAC does not affect the specifications of the other three ANSI RBAC components. The main modification of DS RBAC compared to ANSI RBAC is the extension of the role to a tuple, containing a unique name for the role, and a time-to-live value. Due to this extended role concept in DS RBAC, it is possible to add new flexibility to the RBAC session, while still leaving the major relations and structures of ANSI RBAC untouched.

4.1 Hierarchical RBAC with Dynamic Sessions

There has been much discussion about inheritance in ANSI RBAC. In [Ninghui et al. 2007] it was criticized that the standard was not clear about the role-inheritance semantics. The arguments say, that a role hierarchy of $r_1 \succeq r_2$ would allow three possible interpretations:

- *User Inheritance* (UI): All the users of r_1 are also authorized for r_2. Users "walk down" the inheritance hierarchy.
- *Permission Inheritance* (PI): All the permissions assigned to r_2 are also authorized for r_1. Permissions "walk up" the inheritance hierarchy.
- *Activation Inheritance* (AI): When r_1 is activated in a session, r_2 is also automatically activated in that session. It is not possible however, to access the permissions implied by r_2 without activating r_1.

In [Ferraiolo et al. 2007] the authors of the standard responded to the raised critique, stating that the ANSI RBAC standard treats user and permission inheritance as integrated concepts. This means that role inheritance in ANSI RBAC always involves both user *and* permission inheritance. According to the standard authors, activation inhertiance as introduced by the researchers from Purdue does not exist in ANSI RBAC, as it is possible to activate r_2 independently from r_1 in an inheritance relation of $r_1 \succeq r_2$.

The integration of user and permission inheritance in ANSI RBAC means that the *inheritance relation* is not affected by DS RBAC.

Consider e.g. the simple role hierarchy RH given in figure 2. The solid lines denote the assignment relations and the dotted lines the authorization relations. User u_1 is assigned to role r_1, which has in turn permission p_2 assigned to it. Role r_2 has assigned permission p_2. Hence the relations are as follows: $UA = \{(u_1, r_1)\},\ PA = \{(r_1, p_1), (r_2, p_2)\},\ RH = \{r_1 \succeq r_2\}$. Through the user inheritance relation $authorized\_users(r : ROLES) \rightarrow 2^{USERS}$, u_1 is authorized for role r_2 (The trivial authorization relations $r \succeq r$ are not mentioned, as they are implied by the assignment functions). Due to the permission inheritance relation $authorized\_permissions(r : ROLES) \rightarrow 2^{PRMS}$, role r_1 is authorized for permissions p_2 from role r_2.

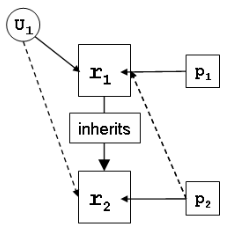

Figure 2: Simple role hierarchy

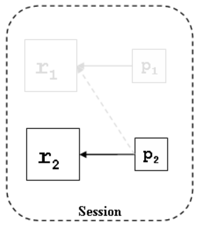

Figure 3: Expiry of role r_1 in activation of both roles of simple role hierarchy

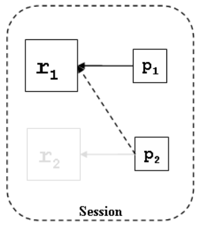

Figure 4: Expiry of role r_2 in activation of both roles of simple role hierarchy

Since the user is authorized for two roles (r_1 and r_2), there are 4 possible role activations in a session: activation of no role, activation of r_1 alone, activation of r_2 alone and activation of both r_1 and r_2.

The first case of no role activation is trivial.

If the user u_1 only activates one role (either r_1 or r_2), the effect of the aging concept of DS RBAC is the same as in the DS RBAC Core model. In case of activation of only r_1, the user would have access to the permissions p_1 and p_2 in the same way as if they had been directly assigned to r_1. In the case of activation of only the inferior role r_2, only the permission p_2, directly assigned to r_2 would be available. Aging would affect the entire role and would only occur if none of the authorized permissions of the activated role was exercised.

Also no effect on the ineritance relation has the activation of both roles r_1 and r_2 by the user. In essence, it is the same situation as if the two roles r_1 and r_2 were not related to each other and just both activated in one session by a user. As figures 3 and 4 show, it is still possible – even in case of a role inheritance relation – that one or more of the roles in the inheritance structure expire.

If e.g. the user u_1 activates both roles r_1 and r_2, but only exercises permission p_2 that is assigned to the inferior role r_2, the superior role r_1 will expire. It is the superior role that will expire, because it is more privileged than r_2. As outlined in the previous section, the $CheckAccess$ function will always identify the least powerful role, according to the well ordering of roles, and update the timestamp of this role. This does not violate the inheritance relation and resambles a desired behavior, as the superior role could e.g. be an administrator role, whereas the inferior role could be an average users role. If the user only accesses user permissions, although she/he has activated also the administrator role, it corresponds to the principle of least privilege, if the non-exercised administrator role expires.

If, on the other hand, the user only exercises the permission p_1 implied by the superior role r_1, the inferior role r_2 will eventually expire, if it has a finite timestamp associated to it (see figure 4). Due to the *authorized_permissions* relation, the permission p_2 orignally assigned to r_2 will still be available to r_1 without a role fault, as r_1 is authorized for all the permissions of its inferior roles.

The aging of a central role in an inheritance hierarchy, does also not adversely affect the inheritance relation. Consider e.g. the inheritance relation depicted in figure 5. As in the previous example, the solid lines denote the assignments (UA, PA) and the dotted lines indicate the authorizations (*authorized_users*, *authorized_permissions*). Figure 6 shows a situation in which the role r_2 that is central to the role hierarchy is no more available in the session. Such a situation could e.g. occur if the user u_1 activates all three possible roles r_1, r_2 and r_3 and r_2 expires. As figure 6 shows, the situation is the same as described in the previous paragraph. The superior role is authorized for all privileges of the inferior roles, no matter whether they are expired or still active. Another important fact is that user u_1 can still directly access or activate role r_3, even if r_2 has been disabled or has expired.

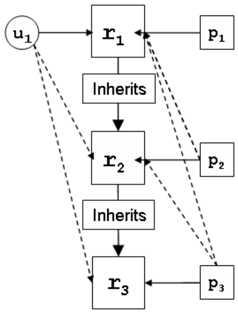

Figure 5: Simple 3-stages role hierarchy

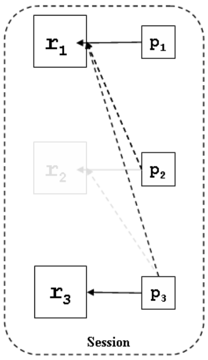

Figure 6: Expiry of central role r_2 in activation of all three roles of simple 3-staged role hierarchy

4.2 Static Separation of Duty (SSD)

As its name implies, Dynamic Sessions in RBAC only affects roles activated in a session. Static Separation of Duty, on the other hand, has to be performed by the system security administrator at design time. Under SSD a user will not even be assigned to conflicting roles, which consequently means that the dynamic aging of roles within a session does not affect this concept.

4.3 Dynamic Separation of Duty (DSD)

In contrast to Static Separation of Duty (SSD), Dynamic Separation of Duty (DSD) occurs on the session level at activation time. The RBAC ANSI standard [ANSI RBAC 2004] defines DSD as collection of pairs $(rs, n) = dsd$ in the set $DSD \subseteq (2^{ROLES} \times N)$, where each rs is a role set and n a natural number $\geq$ 2, with the property that no user may *activate* n or more roles from the given set rs for each $dsd \in DSD$.

As both DSD and DS RBAC affect sessions, one would intuitively expect that DS RBAC affects DSD. But since DS RBAC only takes advantage of the properties from the extended role definition, it does not influence the formal specifications of DSD. When a role expires, it is not removed from the active session. As described in section 3, expiry of a role is only tested in the $CheckAccess$ function, whenever permissions are executed by the user. This consequently means that even if a role that is an element of a role set rs in a dsd, expires, it is not possible for the user to add another active role from the conflicting role set to any of its running sessions, if already the maximum number of $n - 1$ roles from this dsd tuple $((rs, n))$ have been activated.

Consider e.g. a user u_1 who is assigned to the roles r_1, r_2 and r_3 ($UA = \{(u_1, r_1), (u_1, r_2), (u_1, r_3)\}$) and a DSD set defined by the administrator of $DSD = \{(\{r_1, r_2, r_3\}, 2)\}$. This means that the user u_1 can always only activate one role of its 3 role assignments in an active session. If the user u_1 has e.g. activated r_1 in a session and this role expires due to inactivity, the user can still not activate another role from the assigned roles in this very session. The standard demands from the *AddActiveRole* supporting system function in case of DSD that it has to be checked whether the old active session role set ($session\_roles$) added with the new role the user wants to activate, still satisfies the DSD constraint. As the expired role r_1 still is in the role set of the active session ($session\_roles$), by specification of the standard, it is not possible to activate another role.

Consequently the role fault handler would not have to check for a DSD conflict, when re-activating a role, since by the definitions of the ANSI RBAC standard, no DSD conflicts can occur while the role is expired.

5 Related Work

DS RBAC represents a form of automated management of privileges, by automatically adjusting the permission space in a session to those really needed by a user to perform the ongoing tasks.

The idea of automating privilege management in RBAC is not entirely new and has been discussed e.g. in [Sandhu and Bhamidipati 2008, Herzberg et al. 2000, Li et al. 2002, Al-Kahtani and Sandhu 2002, Barka and Sandhu 2000]. The major difference of DS RBAC compared to these models is that DS RBAC exhibits automated privilege selection at the session level and not as an administrative feature. In contrast to the mentioned papers, DS RBAC does not aim at facilitating RBAC administration, by (partly) automating user-role or role-permission assignment. The automatism in DS RBAC rather tries to find the minimum number of permissions (in form of roles) for every initiated session according to the real usage behavior without administrative interference.

Another major difference is that these papers assume that users are willingly adhering to the principle of least privilege, by only choosing the roles necessary to perform certain tasks in a session. We doubt that in practice users will do so. Hence, DS RBAC seeks to limit the available permissions (roles) by inferring the rights needed in the near future from the effective usage behavior of the past.

Automatic privilege management also has been discussed in literature with respect to the user's current location, like e.g. in [Young-Gab and Jongin 2007, Hansen and Oleshchuk 2003, Bertino et al. 2005, Damiani et al. 2007]. While these models are especially valuable for dynamic permission management in mobile environments, they differ significantly from DS RBAC. Privilege management in these models revolves around the user's current position, in contrast to DS RBAC, which takes into account the history of exercised permissions (roles) of an user.

Temporal constraints similar to the time to live used by DS RBAC have also been discussed in RBAC literature e.g. in [Bertino and Bonatti 2001, Joshi et al. 2005, Kyu et al. 2007]. An important difference between these models and DS RBAC is that they propose temporal constraints for fixed, prespecified intervals, whereas in DS RBAC temporal constraints are not tied to a calendar, but rather represent a "sliding window" within which access to objects is allowed.

6 Conclusion

One major drawback of the ANSI RBAC standard is that the activation of roles in a session is left to the users discretion. Users tend to activate more roles and hence hold more privileges than they would need to perform the tasks, for which they have opened the session. This violates the important principle of least privilege.

The proposed model of Dynamic Sessions in RBAC (DS RBAC) seeks to provide a solution to this problem. Its main principle is to dynamically expire roles that are not actively used in a session. Similar to the working set theory in virtual memory, DS RBAC will only keep active the minimum roles containing permissions the user most currently exercised. A role fault occurs whenever the user tries to exercise permissions contained only in expired roles. The role fault handler is then responsible for re-activating the expired role. Optionally, the role fault handler may impose constraints, like e.g. user re-authentication, log-file activation, etc. before re-activating the inactive role. This adds additional security to the model if e.g. an unauthorized user tries to take over an abandoned session. Most importantly, DS RBAC provides a session that tightly sticks to the principle of least privilege, by only keeping those permissions (in the form of roles) active that are effectively exercised by an user.

DS RBAC builds on the specifications provided by the ANSI RBAC standard. As shown in this paper, only limited changes to the standard would be necessary to incorporate the features of DS RBAC.

References

[Al-Kahtani and Sandhu 2002] Al-Kahtani, M., Sandhu, R.: A model for attribute-based user-role assignment. Proceedings 18th Annual Computer Security Applications Conference, 353 – 362, 2002.

[ANSI RBAC 2004] American National Standards Institute, Inc.: American National Stnadard for Inormation Technology - Role Based Access Control (ANSI INCITS 359-2004), 2004.

[Barka and Sandhu 2000] Barka, E., Sandhu, R.: Framework for role-based delegation models. Proceedings of the 16th Annual Computer Security Applications Conference, 2000.

[Barkley 1997] Barkley, J.: Comparing simple role based access control models and access control lists. RBAC '97: Proceedings of the second ACM workshop on Role-based access control, 127–132, New York, NY, USA, 1997. ACM.

[Bertino and Bonatti 2001] Bertino, E., Bonatti, P., Ferrari, E.: TRBAC: A temporal role-based access control model. ACM Trans. Inf. Syst. Secur., 4(3):191–233, 2001.

[Bertino et al. 2005] Bertino, E., Catania, B., Damiani, M., Perlasca, P.: GEO-RBAC: a spatially aware RBAC. SACMAT '05: Proceedings of the tenth ACM symposium on Access control models and technologies, 29–37, New York, NY, USA, 2005. ACM.

[Damiani et al. 2007] Damiani, M., Bertino, E., Perlasca, P.: Data security in location-aware applications: an approach based on RBAC. Int. J. Inf. Comput. Secur., 1(1/2):5–38, 2007.

[Denning 1968] Denning, P.: The Working Set Model for Program Behavior. Communications of the ACM / Volume 11 / Number 5, 323 – 333, May 1968.

[Ferraiolo et al. 2007] Ferraiolo, D., Kuhn, R., Sandhu, R.: RBAC Stanard Rationale - Comments on "A Critique of the ANSI Standard on Role-Based Access Control". IEEE Security & Privacy, 51 – 53, November/December 2007.

[Ferraiolo and Kuhn 1992] Ferraiolo, D., Kuhn, R.: Role-Based Access Control. Proceedings of 15th National Computer Security Conference, 1992, 1992.

[Hansen and Oleshchuk 2003] Hansen, F., Oleshchuk, V.: SRBAC: a spatial role-based access control model for mobile systems. Proceedings of the 7th Nordic Workshop on Secure IT Systems (NORDSEC03). Gjvik, Norway.

[Herzberg et al. 2000] Herzberg, A., Mass, Y., Mihaeli, J., Naor, D. Y. Ravid: Access control meets public key infrastructure, or: Assigning roles to strangers. Proceedings of the 2000 IEEE Symposium on Security and Privacy, 2 – 14, 2000.

[Joshi et al. 2005] Joshi, J., Bertino, E., Usman, L., Ghafoor, A.: A Generalized Temporal Role-Based Access Control Model. IEEE Trans. Knowl. Data Eng., 17(1):4–23, 2005.

[Kyu et al. 2007] Kyu, I., Hyuk, J., Hyun, S., Ung, M.: Context RBAC/MAC Access Control for Ubiquitous Environment . Lecture Notes in Computer Science: Volume 4443/2008, Advances in Databases: Concepts, Systems and Applications, 1075 – 1085, Springer, Berlin / Heidelberg 2007.

[Li et al. 2002] Li, N., Mitchell, J., Winsborough, W.: Design of a role-based trust-management framework. Proceedings IEEE Symposium on Security and Privacy, 114 – 130, 2002.

[Ninghui et al. 2007] Ninghui, L., Byun, J., Bertino, E.: A Critique of the ANSI Standard on Role-Based Access Control. IEEE Security & Privacy, 41 – 49, November/December 2007.

[Osborn et al. 2000] Osborn, S., Sandhu, R., Munawer, Q.: Configuring role-based access control to enforce mandatory and discretionary access control policies. ACM Trans. Inf. Syst. Secur., 3(2):85–106, 2000.

[Sandhu and Bhamidipati 2008] Sandhu, R., Bhamidipati, V.: The ASCAA Principles for Next-Generation Role-Based Access Control. ARES 2008 - International Conference on Availabilityy, Reliability and Security, xxvii – xxxii, 2008.

[Young-Gab and Jongin 2007] Young-Gab, K., Jongin, L.: Dynamic Activation of Role on RBAC for Ubiquitous Applications. ICCIT '07: Proceedings of the 2007 International Conference on Convergence Information Technology, 1148–1153, Washington, DC, USA, 2007. IEEE Computer Society.