The Role Compatibility Security Model

Paper for the Nordic Workshop on Secure IT Systems (NordSec) 2002

Amon Ott
Compuniverse

Email: ao@rsbac.org, WWW: http://www.rsbac.org

October 22, 2002

Abstract

This paper presents the "Role Compatibility" access control model. It has been specially designed to address recent vulnerabilities in network servers by confining compromised services and protecting the base of the system. Furthermore, while being powerful and flexible when needed, it remains fast and easy to use for simple setups.

The model design goals, its specification and implementation outline are presented, followed by a brief comparison to the RBAC and the DTE model. Finally, a Webserver example shows how the model can be used to protect real server systems.

Keywords: Security Model, Access Control, Internet Server, Linux

1 Introduction

As a response to the increasing rate of server vulnerabilities and attacks against them, network server systems require a conceptional solution for better security. Kernel level access control with a specialized security model provides such a solution.

Since no existing model suited our requirements, the Role Compatibility (RC) model has been designed and implemented in the RSBAC framework since December 1998. It supports both a general protection of the base system and an encapsulation of all network service programs to strictly confine security compromises. The abstraction of a role-based model seemed appropriate for this task.

The RSBAC framework provides a generic infrastructure for security model implementations, including persistent list management. It groups access objects into so-called *target types*, e.g. FILE, DIR or IPC[1]. Network access is controlled through *Network Templates*, which provide persistent default attribute values for dynamic network objects.[RSBAC]

The RC model has been in stable production use since January 2000, and a lot of experience with the RSBAC framework and the RC model has been gained. The latest application benchmarks show an RC model overhead of 1.25% against an empty framework, including the Authentication Enforcement (AUTH) module, which is outside the scope of this paper.

As an example, the typical configuration as given in section 6 effectively confines the Apache Webserver and thus prevents an infection by recent Linux malware like the OpenSSL Slapper worm family. The RC model has also been used to secure several Linux firewall configurations, which has been a common DTE model application for some time.[2]

[1] System V Inter Process Communication Object, e.g. Shared Memory.
[2] Section 5 contains a comparison with RBAC and DTE models.

2 Design Goals

The RC model design had to meet most access control requirements on modern Linux based server systems. In detail, the following design goals were accomplished:

Role based model: The abstraction of users to roles and objects to types leads to administration on a functional level and avoids the complexity of per-object control.

Single roles: Each process must have only one current role at a time. Each user ID must have only one default role, which can be assigned to the process when the user ID is acquired.

Program roles: Different programs run by the same user must be able to have different roles. Program roles must override user default roles.

Changing of roles: A process must be able to actively change its current role, if allowed by administration.

Single types: Each object must have only one type.

Granularity: Every role and type combination must have an individual set of allowed accesses.

Separation of administration duty: The model must support separation of duty for administration.

Full configurability: No hard-wired settings should be enforced.

Functional default settings: When unconfigured, the model must allow the system to work as expected.

No changes to existing applications: All applications that are not aware of the model must work as expected, unless when they have insufficient privileges.

Adaptive complexity: Model administration should be only as complex as necessary to meet the actual requirements. Using default settings, the model must behave as a simple role model. All special behaviour must be optional.

3 Specification

3.1 Basic Definitions

Within the RC model specification, the active entities (subjects) are processes working on behalf of users and executing one program file with a set of dynamic libraries at a time.

Objects are grouped into the RSBAC framework target types, but different groupings of objects would not change the model significantly.

Access rights are the standard framework request types plus some model specific rights. Like modified object groupings, a different set of standard access rights would not affect the model itself.

The following terms will be used:

- owner(p:process):user := owner of process p
- parent(p:process):process := parent process of process p
- program(p:process):file := program file currently executed by process p
- parent(f:filesystem object):filesystem object := parent object of filesystem object f
- $attributename_{tn}$(o:object):valuetype := value of attribute *attributename* of object o at time n

Processes as subjects can perform some model relevant actions:

- changeowner$_{tn}$(p:process, u:user) := change owner of process p to u at time n
- clone$_{tn}$(p_1:process, p_2:process) := creation of process p_2 by parent process p_1 at time n
- execute$_{tn}$(p:process, f:file) := start execution of program file f in process p at time n
- createfs$_{tn}$(p:process, f:filesystem object) := creation of filesystem object f by process p at time n
- createipc$_{tn}$(p:process, i:IPC object) := creation of IPC object i by process p at time n

Three types of rules will be specified:

1. Invariants define rules, which must always be met. Here the effective values of inheritable filesystem object attributes are determined.
2. Transitions define the next state of an attribute after a certain action.
3. Constraints define the conditions to be met when an action is performed.

3.2 Roles and Types

3.2.1 Roles

Roles and types are identified by a non-negative integer index. Every process has one current role, and every user has one default role.

- currentrole$_{tn}$(p:process):role := current role of process p at time n
- defrole$_{tn}$(u:user):role := default role of user u at time n

The first system process 0 gets the default role of user 0.

$$\text{currentrole}_{t0}(0) := \text{defrole}_{t0}(0) \tag{1}$$

In the default setting, i.e., if not specified differently by the *Initial Role* or the *Forced Role* attributes of the program file or the process (see subsection 3.5), the following implicit role transitions are performed:

- All subprocesses inherit the current role of their parent process.

$$\text{clone}_{tn}(p_1, p_2) \Rightarrow \text{currentrole}_{tn+1}(p_2) := \text{currentrole}_{tn}(p_1) \tag{2}$$

- On every change of the process owner, the process current role is set to the new owner's default role.

$$\text{changeowner}_{tn}(p, u) \Rightarrow \text{currentrole}_{tn+1}(p) := \text{defrole}_{tn}(u) \tag{3}$$

- When another program is executed, the current role of the process is kept.

$$\text{execute}_{tn}(p, f) \Rightarrow \text{currentrole}_{tn+1}(p) := \text{currentrole}_{tn}(p) \tag{4}$$

Thus, the default scheme follows the expected behaviour of a role based model.

3.2.2 Types

Every object has an RC type. Hierarchically organized objects of RSBAC target types FILE, DIR, FIFO and SYMLINK[3] can have a special type value *inherit parent*, in which case the parent object's type is used. If there is no parent, the default value 0 is applied.

Whenever values may be inherited, the term *effective value* is used to denote the final value. Inheritance greatly reduces the number of attribute values to be stored and follows the usual way of grouping objects in hierarchies.

- type_{tn}(o:object):type := type of object o at time n
- efftype_{tn}(f:filesystem object):type := effective type of filesystem object f at time n, including inheritance

The effective type is derived as follows:

$$\text{efftype}_{tn}(\text{f}) := \begin{cases} \text{if } \text{type}_{tn}(\text{f}) = \text{inherit\_parent} \wedge \exists \text{parent}(\text{f}) : \text{efftype}_{tn}(\text{parent}(\text{f})) \\ \text{if } \text{type}_{tn}(\text{f}) = \text{inherit\_parent} \wedge \nexists \text{parent}(\text{f}) : 0 \\ \text{if } \text{type}_{tn}(\text{f}) \neq \text{inherit\_parent} : \text{type}_{tn}(\text{f}) \end{cases} \quad (5)$$

When a new filesystem object is created, its type is set to the value of the role attribute *Default fd create type* of the current role of the creating process, which can also be the special value *inherit parent* mentioned above.

$$\text{createfs}_{tn}(\text{p}, \text{f}) \Rightarrow \ \text{type}_{tn+1}(\text{f}) := \text{default\_fd\_create\_type}_{tn}(\text{currentrole}_{tn}(\text{p})) \quad (6)$$

When a new process object is created, its type is set depending on the value of the role attribute *Default process create type* of the current role of the creating process. The special and default value *inherit parent* sets the type value to that of the creating process. Be

$$\text{pct}_{tn}(\text{p}_1) := \text{default\_process\_create\_type}_{tn}(\text{currentrole}_{tn}(\text{p}_1)) \quad (7)$$

$$\text{clone}_{tn}(\text{p}_1, \text{p}_2) \Rightarrow \ \text{type}_{tn+1}(\text{p}_2) := \begin{cases} \text{if } \text{pct}_{tn}(\text{p}_1) = \text{inherit\_parent} : \text{type}_{tn}(\text{p}_1) \\ \text{if } \text{pct}_{tn}(\text{p}_1) \neq \text{inherit\_parent} : \text{pct}_{tn}(\text{p}_1) \end{cases} \quad (8)$$

On execution of a new program file, the process type is set according to the value of the role attribute *Default process execute type* of the current role of the process. The special and default value *inherit parent* leaves the process type unchanged. Be

$$\text{pet}_{tn}(\text{p}) := \text{default\_process\_execute\_type}_{tn}(\text{currentrole}_{tn}(\text{p})) \quad (9)$$

$$\text{execute}_{tn}(\text{p}, \text{f}) \Rightarrow \ \text{type}_{tn+1}(\text{p}) := \begin{cases} \text{if } \text{pet}_{tn}(\text{p}) = \text{inherit\_parent} : \text{type}_{tn}(\text{p}) \\ \text{if } \text{pet}_{tn}(\text{p}) \neq \text{inherit\_parent} : \text{pet}_{tn}(\text{p}) \end{cases} \quad (10)$$

Changing the owner of a process leads to the process type being set to the value of the role attribute *Default process chown type* of the current role of the process. The special and default value *inherit parent* leaves the process type unchanged. The other valid special value *use new role def create* uses the value of the role attribute *Default process create type* of the new current role of the process (see Roles). Be

$$\text{pot}_{tn}(\text{p}) := \text{default\_process\_chown\_type}_{tn}(\text{currentrole}_{tn}(\text{p})) \quad (11)$$

$$\text{pct}_{tn+1}(\text{p}) := \text{default\_process\_create\_type}_{tn}(\text{currentrole}_{tn+1}(\text{p})) \quad (12)$$

$$\text{changeowner}_{tn}(\text{p}, \text{u}) \Rightarrow$$

$$\text{type}_{tn+1}(\text{p}) := \begin{cases} \text{if } \text{pot}_{tn}(\text{p}) = \text{inherit\_parent} : \text{type}_{tn}(\text{p}) \\ \text{if } \text{pot}_{tn}(\text{p}) = \text{use\_new\_role\_def\_create} : \text{pct}_{tn+1}(\text{p}) \\ \textit{else} : \text{pot}_{tn}(\text{p}) \end{cases} \quad (13)$$

[3]Objects of these target types are also referenced as *filesystem objects*.

Finally, the types of newly created IPC objects can be influenced by the value of the role attribute *Default ipc create type* of the current role of the process.

$$\text{createipc}_{tn}(\text{p}, \text{i}) \Rightarrow \ \text{type}_{tn+1}(\text{i}) := \text{default\_ipc\_create\_type}_{tn}(\text{currentrole}_{tn}(\text{p})) \tag{14}$$

The types of all newly created network objects are derived from their templates[4] and cannot be preset through role attributes.

Default type values provide a mandatory way to keep new objects suitable for the roles that created them, while completely avoiding discretionary elements for type selection and the necessity of making applications aware of the access control model.

3.3 Role Compatibility

While most changes of the current role of a process are implicit with certain actions, processes can also actively change their current role. This is specially useful for short term administration tasks and for server programs, whose subprocesses have to act in several roles without changing their user ID.[5] In both cases, the original role should not be regained.

- changerole_{tn}(p:process, r:role) := process p actively changes its current role to r at time n

The right to do so is called *Role Compatibility*: A process may change its current role r_1 to role r_2, if role r_2 is in the set of compatible roles of role r_1.

- comproles(r:role):set of roles := set of compatible roles for role t

$$\text{changerole}_{tn}(\text{p}, \text{r}) \Rightarrow \ r \in \text{comproles}_{tn}(\text{currentrole}_{tn}(\text{p})) \tag{15}$$

3.4 Type Compatibility

Accesses by processes performing a current role to objects of certain types are controlled through the *Type Compatibility* settings.

- getaccess_{tn}(p:process, o:object, a:access_type) := process p gets access to object o with access type (request type) a at time t

A process with current role r may access objects of type t with accesses of access type (request type) a, if role r is marked as *Type Compatible* with type t for access type a.

- compatible_{tn}(r:role, t:type, a:access_type) := role r is marked as compatible with type t for access type (request type) a at time n

$$\text{getaccess}_{tn}(\text{p}, \text{o}, \text{a}) \Rightarrow \ \text{compatible}_{tn}(\text{currentrole}_{tn}(\text{p}), \text{efftype}_{tn}(\text{o}), \text{a}) \tag{16}$$

Type compatibility sets are kept separately for the different RSBAC target types.

3.5 Program Based Roles with Initial and Forced Roles

There are two ways to assign roles to programs: initial and forced roles. Both program role settings are kept as file attributes.

[4] RSBAC Network Templates, see documentation at [RSBAC].

[5] See Webserver example in section 6.

3.5.1 Initial Roles

If an initial role has been assigned to a program file, it is set as current role of every process that executes this program. However, the role can be changed at any time by all implicit or explicit mechanisms mentioned above, e.g. by changing the process owner. Initial roles are typically used for login programs, which need special privileges for authentication, but have to switch to a new owner's default role afterwards.

Two special initial role values affect implicit role transitions:

role_inherit_parent (default value): Get initial role setting from filesystem parent object. If there is no parent object, use root dir default value role_use_forced_role. This default value allows to set an initial role for whole directory trees.

role_use_forced_role (root dir default value): Only use the forced role setting.

As usual, the inheritance implies the notion of effective values:

- initialrole$_{tn}$(f:file):role := initial role value of file f at time t
- effinitialrole$_{tn}$(f:file):role := effective initial role of file f at time n, including inheritance from parent filesystem objects

The effective initial role is derived as follows:

$$\begin{aligned}&\text{effinitialrole}_{tn}(\text{f}) :=\\ &\quad\begin{cases}\text{if initialrole}_{tn}(\text{f}) = \text{inherit\_parent} \ \wedge \exists \text{parent(f)} : \text{effinitialrole}_{tn}(\text{parent(f)})\\ \text{if initialrole}_{tn}(\text{f}) = \text{inherit\_parent} \ \wedge \ \nexists \text{parent(f)} : \text{role\_use\_forced\_role}\\ \qquad\qquad \text{if initialrole}_{tn}(\text{f}) \neq \text{inherit\_parent} : \text{initialrole}_{tn}(\text{f})\end{cases}\end{aligned} \tag{17}$$

Initial roles for program files change the implicit role transition on execution from rule 4 as follows:

$$\begin{aligned}&\text{execute}_{tn}(\text{p},\text{f}) \Rightarrow \ \text{currentrole}_{tn+1}(\text{p}) :=\\ &\quad\begin{cases}\text{if effinitialrole}_{tn}(\text{f}) = \text{role\_use\_forced\_role} : \mathit{follow\ rule}\ 20\\ \text{if effinitialrole}_{tn}(\text{f}) \neq \text{role\_use\_forced\_role} : \text{effinitialrole}_{tn}(\text{f})\end{cases}\end{aligned} \tag{18}$$

3.5.2 Forced Roles

While initial roles are only set as temporary current roles, forced roles are kept until either another program with initial or forced role is executed or the process actively changes to a compatible role. Certainly, it has to be kept in a process attribute for later use.

All other implicit mechanisms, e.g. when changing the process owner, do not affect the current role while a forced role is set.

Forced roles are useful for those server program encapsulation cases, where a server program must always run with the same privileges for all process owners.

There are several special forced role values which affect implicit role transitions:

role_inherit_user: Always set the (new) process owner's default role as current role when executing this program or when changing process owner while this program is executed. This can be used for login shells to make sure that the user's default role is used.

role_inherit_process: Keep the current role when executing this program or changing process owner while this program is executed. This value lets subprograms keep the forced role of their parents in all cases.

role_inherit_parent (default value): Get forced role setting from filesystem parent object. If there is no parent object, use root dir default value role_inherit_up_mixed. This default value allows to set a forced role for whole directory trees.

role_inherit_up_mixed (root dir default value): Keep the current role when executing this program, but set it to new owner's default role when changing the process owner. This is the standard role model behaviour as mentioned above.

The forced role default settings make all programs run with the process owner's default role, which is the desired behaviour in most cases.

- forcedrole_{tn}(f:file):role := forced role value set for file f at time n
- $\text{effforcedrole}_{tn}$(f:file):role := effective forced role of file f at time n, including inheritance from parent filesystem objects

The effective forced role is derived as follows:

$$\text{effforcedrole}_{tn}(\text{f}) := \begin{cases} \text{if forcedrole}_{tn}(\text{f}) = \text{inherit\_parent} \ \wedge \exists \text{parent(f)} : \text{effforcedrole}_{tn}(\text{parent(f)}) \\ \text{if forcedrole}_{tn}(\text{f}) = \text{inherit\_parent} \wedge \nexists \text{parent(f)} : \text{role\_inherit\_up\_mixed} \\ \text{if forcedrole}_{tn}(\text{f}) \neq \text{inherit\_parent} : \text{forcedrole}_{tn}(\text{f}) \end{cases} \tag{19}$$

Forced roles for program files extend the implicit role transition on execution from rule 18 as follows:[6]

$$\text{execute}_{tn}(\text{p},\text{f}) \Rightarrow \ \text{currentrole}_{tn+1}(\text{p}) := \begin{cases} \text{if effforcedrole}_{tn}(\text{f}) = \text{role\_inherit\_user} : \text{defrole}_{tn}(\text{owner(p)}) \\ \text{if effforcedrole}_{tn}(\text{f}) = \text{role\_inherit\_process} : \text{currentrole}_{tn}(\text{p}) \\ \text{if effforcedrole}_{tn}(\text{f}) = \text{role\_inherit\_up\_mixed} : \text{currentrole}_{tn}(\text{p}) \\ else : \text{effforcedrole}_{tn}(\text{f}) \end{cases} \tag{20}$$

The effective forced role value from the executed file is copied to the respective process attribute.

$$\text{execute}_{tn}(\text{p},\text{f}) \Rightarrow \ \text{forcedrole}_{tn+1}(\text{p}) := \text{effforcedrole}_{tn}(\text{f}) \tag{21}$$

The implicit role transition on process owner changes from rule 3 is modified as well:

$$\text{changeowner}_{tn}(\text{p},\text{u}) \Rightarrow \ \text{currentrole}_{tn+1}(\text{p}) := \begin{cases} \text{if forcedrole}_{tn}(\text{p}) = \text{role\_inherit\_user} : \text{defrole}_{tn}(\text{u}) \\ \text{if forcedrole}_{tn}(\text{p}) = \text{role\_inherit\_process} : \text{currentrole}_{tn}(\text{p}) \\ \text{if forcedrole}_{tn}(\text{p}) = \text{role\_inherit\_up\_mixed} : \text{defrole}_{tn}(\text{u}) \\ else : \text{forcedrole}_{tn}(\text{p}) \end{cases} \tag{22}$$

3.6 Standard Administration

In the standard case, all RC model administration is done through one or more roles, which have their *Admin Type* attribute set to *Role Admin*. This value gives full administrative privileges, overriding the separation scheme presented in the next subsection, but no access rights to objects.

Administration tasks are definition of roles and types, specification of compatibilities, assignment of default, initial and forced roles to users and program files and assignment of types to objects.

3.7 Separation of Administration Duty

Security administration should best be separated into several tasks, performed by several different administrators, which have to cooperate to provide additional privileges.

The Role Compatibility Model contains a separation of administration duty scheme, which allows to generate limited workgroups as well as enforce cooperation of two or more roles for

[6] If the initial role is set to the default value role_use_forced_role.

most administration tasks. However, the *Admin Type* role attribute makes the separation scheme completely optional.

As the separation of duty related settings can only be changed by roles with *Admin Type* set to *Role Admin*, removing these roles or resetting their *Admin Type* value fixes the separation for future use.

3.7.1 Admin Roles

Every role definition contains a set of roles, called *Admin Roles*, which processes performing this role are allowed to administrate. For many settings, e.g. the compatibility sets, additional privileges are required, which are explained below.

- adminroles(r:role):set of roles := set of administrated roles for role r
- $\text{administraterole}_{tn}$(p:process, r:role) := process p administrates settings of role r at time n

$$\text{administraterole}_{tn}(\text{p},\text{r}) \Rightarrow \; r \in \text{adminroles}_{tn}(\text{currentrole}_{tn}(\text{p})) \tag{23}$$

The *Admin Roles* set of any role can only be changed by roles with *Admin Type* value *Role Admin*.

- $\text{changeadminroles}_{tn}$(p:process, r:role) := process p changes the set of admin roles of role r at time n

$$\text{changeadminroles}_{tn}(\text{p},\text{r}) \Rightarrow \; \text{admintype}_{tn}(\text{currentrole}_{tn}(\text{p})) = \text{role\_admin} \tag{24}$$

3.7.2 Assign Roles

Another set of roles contained in all role definitions is called *Assign Roles*. It defines, which roles processes running this certain role are allowed to assign as compatible role to roles, as default role to users or as initial or forced role to program files or processes.

- assignroles(r:role):set of roles := set of assignable roles for role r
- addcomprole_{tn}(p:process, r_1:role, r_2:role) := process p adds role r_1 to the set of compatible roles of role r_2 at time n
- $\text{assigndefrole}_{tn}$(p:process, r:role, u:user) := process p assigns default role r to user u at time n
- $\text{assigninitialrole}_{tn}$(p:process, r:role, f:file) := process p assigns initial role r to program file f at time n
- $\text{assignforcedrole}_{tn}$(p:process, r:role, f:file) := process p assigns forced role r to program file f at time n
- $\text{assignforcedrole}_{tn}$($\text{p}_1$:process, r:role, p_2:process) := process p_1 assigns forced role r to process p_2 at time n

$$\begin{aligned} \text{addcomprole}_{tn}(\text{p},\text{r}_1,\text{r}_2) \quad &\Rightarrow \quad \text{r}_1 \in \text{assignroles}_{tn}(\text{currentrole}_{tn}(\text{p})) \\ &\wedge \quad \text{r}_2 \in \text{adminroles}_{tn}(\text{currentrole}_{tn}(\text{p})) \end{aligned} \tag{25}$$

Default roles can only be assigned to users, if both the old and the new role are in the set of *Assign Roles*. This restriction, together with the sets of compatible roles, creates a range of reachable roles, which easily forms a workgroup.

$$\text{assigndefrole}_{tn}(\text{p},\text{r},\text{u}) \Rightarrow \; \text{r}, \text{defrole}_{tn}(\text{u}) \in \text{assignroles}_{tn}(\text{currentrole}_{tn}(\text{p})) \tag{26}$$

To set an initial or forced role for a program file or process object, the additional right MODIFY_ATTRIBUTE to the type of the object is needed.

$$\begin{aligned}&\text{assigninitialrole}_{tn}(\mathrm{p},\mathrm{r},\mathrm{f}) \Rightarrow \\ &\quad r \in \text{assignroles}_{tn}(\text{currentrole}_{tn}(\mathrm{p})) \\ &\wedge \quad \text{compatible}_{tn}(\text{currentrole}_{tn}(\mathrm{p}), \text{efftype}_{tn}(\mathrm{f}), \text{MODIFY\_ATTRIBUTE})\end{aligned} \tag{27}$$

$$\begin{aligned}&\text{assignforcedrole}_{tn}(\mathrm{p},\mathrm{r},\mathrm{f}) \Rightarrow \\ &\quad \mathrm{r} \in \text{assignroles}_{tn}(\text{currentrole}_{tn}(\mathrm{p})) \\ &\wedge \quad \text{compatible}_{tn}(\text{currentrole}_{tn}(\mathrm{p}), \text{efftype}_{tn}(\mathrm{f}), \text{MODIFY\_ATTRIBUTE})\end{aligned} \tag{28}$$

$$\begin{aligned}&\text{assignforcedrole}_{tn}(\mathrm{p}_1,\mathrm{r},\mathrm{p}_2) \Rightarrow \\ &\quad \mathrm{r} \in \text{assignroles}_{tn}(\text{currentrole}_{tn}(\mathrm{p}_1)) \\ &\wedge \quad \text{compatible}_{tn}(\text{currentrole}_{tn}(\mathrm{p}_1), \text{type}_{tn}(\mathrm{p}_2), \text{MODIFY\_ATTRIBUTE})\end{aligned} \tag{29}$$

Changes to the *Assign Roles* set of any role are restricted to roles with *Admin Type* value *Role Admin*.

- changeassignroles$_{tn}$(p:process, r:role) := process p changes the set of assign roles of role r at time n

$$\text{changeassignroles}_{tn}(\mathrm{p},\mathrm{r}) \Rightarrow \text{admintype}_{tn}(\text{currentrole}_{tn}(\mathrm{p})) = \text{role\_admin} \tag{30}$$

3.7.3 Special Rights

Some special rights to types have been defined:

ADMIN: Administrate this type, i.e., change type name or remove type.

ASSIGN: Assign this type to objects. Additionally, MODIFY_ATTRIBUTE to the previous type of the object is needed.

ACCESS_CONTROL: Change type compatibility settings for this type and all requests, which are no special rights.

SUPERVISOR: Change type compatibility settings for this type for all special rights. If no role has SUPERVISOR right or *Admin Type* set to *Role Admin*, the special right settings can no longer be changed.

- specialrights := {ADMIN, ASSIGN, ACCESS_CONTROL, SUPERVISOR}
- administratetype$_{tn}$(p:process, t:type) := process p administrates type t at time n
- assigntype$_{tn}$(p:process, t:type, o:object) := process p assigns the type t to object o at time n
- changetypecomp$_{tn}$(p:process, r:role, t:type, a:access_type) := process p adds or removes access type a to or from the type compatibility set of role r to type t at time n

$$\begin{aligned}&\text{administratetype}_{tn}(\mathrm{p},\mathrm{t}) \Rightarrow \\ &\quad \text{compatible}_{tn}(\text{currentrole}_{tn}(\mathrm{p}), \mathrm{t}, \text{ADMIN})\end{aligned} \tag{31}$$

$$\begin{aligned}&\text{assigntype}_{tn}(\mathrm{p},\mathrm{t},\mathrm{o}) \Rightarrow \\ &\quad \text{compatible}_{tn}(\text{currentrole}_{tn}(\mathrm{p}), \mathrm{t}, \text{ASSIGN})\end{aligned}$$

$$\wedge \quad \text{compatible}_{tn}(\text{currentrole}_{tn}(\text{p}), \text{efftype}_{tn}(\text{o}), \text{MODIFY\_ATTRIBUTE}) \tag{32}$$

$$\begin{aligned} &\text{changetypecomp}_{tn}(\text{p}, \text{r}, \text{t}, \text{a}) \wedge \ \text{a} \notin \text{specialrights} \Rightarrow \\ &\qquad \text{compatible}_{tn}(\text{currentrole}_{tn}(\text{p}), \text{t}, \text{ACCESS\_CONTROL}) \\ &\wedge \quad \text{r} \in \ \text{adminroles}(\text{currentrole}_{tn}(\text{p})) \end{aligned} \tag{33}$$

$$\begin{aligned} &\text{changetypecomp}_{tn}(\text{p}, \text{r}, \text{t}, \text{a}) \wedge \ \text{a} \in \text{specialrights} \Rightarrow \\ &\qquad \text{compatible}_{tn}(\text{currentrole}_{tn}(\text{p}), \text{t}, \text{SUPERVISOR}) \\ &\wedge \quad \text{r} \in \ \text{adminroles}(\text{currentrole}_{tn}(\text{p})) \end{aligned} \tag{34}$$

3.8 Lifetime Limits

All compatibility, *Admin Roles* and *Assign Roles* settings have an optional time-to-live parameter. After the given time, the set of requests for a type compatibility setting gets cleared by the system, while compatible, admin or assign roles get removed from their set.

With lifetime limits, temporary additional rights do not require manual action to be revoked and thus avoid this typical situation of unnecessary permanent rights. Certainly, time-to-live settings rely on the correct system time to be always maintained.

4 Implementation

The Role Compatibility model has been implemented as a decision module for the RSBAC framework[RSBAC] and makes extensive use of its infrastructure.

The Rule Set Based Access Control (RSBAC) system is an open source security extension to current Linux kernels, which has been continuously developed by the author for several years.

RSBAC was designed according to the Generalized Framework for Access Control (GFAC)[Abrams+90] to overcome the deficiencies of access control in standard Linux systems, and to make a flexible combination of security models as well as proper access logging possible.

Only smaller RC changes and adaptions to changes of the framework have been made from November 1999 till November 2001, like initial roles or the extension for new target types. From November 2001, the RC model implementation has been moved to generic RSBAC lists and the original limit of 64 roles and 64 RC types per target type has been removed. Also, the new network target types and time limits have been included.

4.1 Roles and Types

Role and type definitions are registered as persistent generic lists with their index number as list index.

Role data includes a role name for human use and the simple attributes *Admin Type*, *Default fd create type*, *Default process create type*, *Default process chown type*, *Default process execute type* and *Default IPC create type*.

Type data only includes the type name for human use.

4.2 Role Compatibility, Admin and Assign Roles

Persistent generic lists of lists without data are registered for Role Compatibility, Admin and Assign Roles. The first level index is the role number of the set owner, the second level index the role number of the set member.

Set membership is tested by existence of the second level entry.

4.3 Type Compatibility

For each RSBAC target type one persistent list of lists is registered. The first level index is the role number, the second level index is the type number. Only second level data is used, it contains the set of allowed requests coded as a 64 bit integer used as bit set.

Absence of an item is interpreted as the default value of an empty set.

4.4 Program Based Access Control with Initial and Forced Roles

Unlike the above items, initial and forced role settings are implemented as attributes of file objects, the forced role also as attribute of process objects. They are kept and provided by the RSBAC General Data Structures component.

4.5 Access Control Decision and Notification

RSBAC request decisions and respective automatic attribute updates are performed in the decision function rsbac_adf_request_rc and the notification function rsbac_adf_set_attr_rc, which are called from the ADF dispatcher functions.[7]

Administration and role changing decisions are made in the respective individual functions, which have been implemented as additional system calls.

For most requests, the decision function only takes the process current role, the object type and the request and matches them against the type compatibility settings.

The notification function performs all implicit role and type changes for existing or newly created processes and objects as specified above.

The time values *tn* and *tn+1* used in the specification are interpreted as *at the time of the decision request call* and *directly after the notification call*. Since all attribute changes from the specification rules are either for the requesting process only, or for a newly created object, which cannot be accessed by any process before notification has completed, race conditions can only occur with active administration.

5 Comparison with RBAC and DTE Models

5.1 Role Based Access Control (RBAC)

5.1.1 Model Description

The RBAC access control model as described in [FerKuh92] defines subjects, roles and transactions. A transaction is defined as a transformation procedure plus its necessary data accesses. All subject activities in a system are performed through transactions, but not the system tasks like identification or authentication.

The RBAC model defines three basic rules:

1. Role assignment: A subject can execute a transaction only if the subject has selected or been assigned a role.
2. Role authorization: A subject's active role must be authorized for the subject.
3. Transaction authorization: A subject can execute a transaction only, if the transaction is authorized for the subject's active role.

Additionally, transformation procedures, objects and access modes can be separated, and an access function can define, which role executing which transaction may access which objects with which access modes.

In [FeCuKu95], the term *operation* is introduced, which denotes an access with a certain mode to a set of objects. Roles are then authorized for operations and no longer for transactions or

[7] See e.g. [RSBAC, Ott2001, Ott2001a] for RSBAC structure.

transaction procedures. Also, users are distinguished from subjects. A subject is an active entity, performing operations on behalf of one user at a time, and has a set of active roles, for which the user must be authorized.

Roles may be members of other roles, so that membership in a subrole implies the membership in all parent roles, including all their authorizations. The possible membership in several roles requires the definition of *mutual exclusion* to preserve separation of duty, i.e., pairs of roles which may not share the same member or, in the revised model, which may not be activated at the same time by the same subject.

Finally, the RBAC model defines static and dynamic *capacities* of roles, the first being the maximum number of members, the latter the maximum number of subjects having the role activated.

In [Ferraiolo+2001], a NIST standard for RBAC models has been proposed. It adds the notion of *user sessions*, which allow to selectively activate or deactivate roles within a session. All RBAC features are grouped into *Core RBAC*, which contains the basic functionality, *Hierarchical RBAC* to define role hierarchies and *Constrained RBAC* with *Static* and *Dynamic Separation of Duty Relations*. All RBAC separation of duty relates to what roles from the assigned set of roles can be used by a single user at the same time. Of this, mutual exclusion is only a subset.

5.1.2 Comparison to RC Model

Similar to the RBAC model, RC defines subjects as processes, the active entities within a system, working on behalf of users with a current role and performing accesses to objects. However, in RC model each process can only have one active role at a time, avoiding the complex scheme of mutual exclusion. This means that in some cases several roles with overlapping rights may have to be defined.

The RBAC set of authorized roles of users is covered by the RC set of compatible roles, which are reachable from the user's default role. Even more, after changing into a role there might not be a way back to the original role. This can effectively avoid uncontrolled flow of information through process memory by switching to another role with higher privileges and then back to the original role.

The RC model can even simulate the transaction concept from the first RBAC version through program based roles and separate types for the program files: Transaction authorization is mapped as EXECUTE right on the program object type, while operations allowed for a transaction can be assigned as compatibilities to the program's assigned role.

The RBAC model does not have an equivalent for RC type abstraction, program based roles and separation of administration duty. Time limits can only be simulated through dynamic mutual exclusion.

RC only lacks role capacities, which were not considered as useful.

5.2 Domain and Type Enforcement (DTE)

5.2.1 Model Description

As stated in [Badger+95], Domain and Type Enforcement is based on an enhanced version of Type Enforcement (TE). The main additions to the original model are a high level policy specification language and a human readable format of attribute values in the runtime policy database.

Type Enforcement is a table based access control model. Active entities, the subjects, have an attribute *Domain*, while passive entities, the objects, have a *type* attribute. Possible accesses by subjects to objects are grouped into the *access modes* read, write, execute and traverse.

A global *Domain Definition Table (DDT)* contains the allowed interactions, where domains and types form rows and columns, and each cell holds a set of access modes.

Subject-to-subject access control is based on a global *Domain Interaction Table (DIT)* with subjects as both descriptors and, again, a set of access modes, e.g. signal, create or destroy, in the cells.

In contrast to the original TE model, DTE supports *implicit* attribute maintenance. This means that values may be only kept on a higher level of the directory and file hierarchy, but are used for all levels below as well. Also, the specification language allows to specify types by lookup path prefixes.

The first process on a system, the init process, gets a predefined initial domain assigned. Each process can enter another domain by executing a program bound to it, a so-called *entry point*. An entry point may be executed to explicitely enter one of its associated domains, if the subject's current domain has *exec* right on the target domain. The *auto* access right to a domain automatically selects this domain, if one of its entry points gets executed.

The user-domain relationship is entirely built on entry points like command shells etc. However, a DTE aware login program can select from all domains associated with an entry point to avoid individual copies for each domain.

5.2.2 Comparison to RC Model

While the RC model makes role assignments based on users and programs, both represented by processes, the DTE model itself avoids the concept of users and only focuses on programs. User representation and role assignment are placed under the discretion of unspecific DTE aware applications outside the scope of the model.

Another DTE drawback is that roles can only be changed through entry point programs, while the RC model allows to dynamically switch to compatible roles within one single application and to default roles on every change of the process owner. Dynamic role changes are specially useful for user based server programs.

Finally, DTE administration concepts were not mentioned in [Badger+95] and thus remain unclear.

6 Application Example

This section describes how the RC model is used to secure a typical server system. The approach given here can easily be adapted to many other types of services.

6.1 Base Protection

Those parts of the RSBAC and RC access control setup, which are applicable for all types of systems, are called *Base Protection*. Objects to be protected include the basic directory structure, executables, libraries, configuration files, kernel objects and boot loaders, raw devices, account and authentication data, log files, home directories etc.

Generally, for each of these object categories one RC type gets defined and assigned to the individual objects, and all existing roles get appropiate type compatibility settings to these types.

As an example, the type *Executables* is assigned to the directories /bin and /usr/bin, which contain executable files. All of these then inherit the effective type *Executables*. As soon as the type compatibilities of all roles to this type are set accordingly, executables are fully protected. Furthermore, after setting *all* desired executables to this type, one can safely remove the execute right to all other types and thus avoid any execution of unprotected and possibly malicious files.

6.2 Service Encapsulation

While the Base Protection secures the base system, the different services are additionally encapsulated to restrict them to the absolutely necessary. Here program based roles are most useful.

A typical example of good RC usage is a virtual Webserver system with an arbitrary number of customers, who want to use their own CGI scripts with private data.

We use a forced role *Webserver*, which gets assigned to the Webserver binary. This role may not access any of the base protection types except the mapping of libraries. The general Webserver logging type *Webserver Log* can be accessed to create and append to log files.

Each customer C gets a separate directory tree, three RC types, called *Web-Data-C*, *CGI-Program-C* and *Private-Data-C*, and three roles, called *Webserver-C*, *Upload-C* and *CGI-C*.

The general Webserver role may not access any customer data. Instead, a serving process changes to the compatible role Webserver-C when serving content for customer C. The role handling can e.g. be implemented in a simple Apache module.

Role Webserver-C may read Web-Data-C and execute CGI-Program-C. The CGI folder for customer C has a forced role setting of CGI-C, which gets inherited to all programs in it. Thus, when one of C's CGI programs is run, it uses role CGI-C and gets limited access to all of C's data, specially Private-Data-C.

Finally, the upload account for customer C gets the default role Upload-C, which has read and write, but no execute access to all three types. Access to any other type is denied.

6.3 Further Refinement

The setup presented here makes use of several RC features like compatible roles and program based roles to protect the system and the customers from each other and from client systems.

It can easily be extended by network access control to prevent unwanted connections by customer CGIs or a compromised server program, e.g. to avoid the spreading of worms.

For a complete setup, every single service can be encapsulated with individual roles and types. The focus should certainly lie on the network services.

7 Conclusion

Practical experience with server systems using the Role Compatibility model for access control shows that base protection and service encapsulation are possible without drawbacks in usability. All protection requirements of these systems could be solved by proper RC configuration, while the well-known RBAC and DTE models each show several deficiencies.

The RC model as presented in this paper proved to be easy to use in simple setups, but also very flexible and powerful in complex environments. Combined with the RSBAC concept of Network Templates, even access to and from remote systems can be effectively controlled.

References

[Abrams+90] Abrams, M. D., Eggers, K. W., La Padula, L. J., Olson, I. M., A Generalized Framework for Access Control: An Informal Description, Proceedings of the 13th National Computer Security Conference, Oktober 1990

[Badger+95] Badger, L., Sterne, D. F., Sherman, D. L., Walker, K. M., Haghighat, S. A., Practical Domain and Type Enforcement for UNIX, 1995 IEEE Symposium on Security and Privacy

[FerKuh92] Ferraiolo, D., Kuhn, R., Role-Based Access Control, Proceedings of the 15th National Computer Security Conference, 1992

[FeCuKu95] Ferraiolo, D. F., Cugini, J. A., Kuhn, D. R., Role-Based Access Control (RBAC): Features and Motivations, Proceedings of the Computer Security Applications Conference 1995

[Ferraiolo+2001] Ferraiolo, D. F., Sandhu, R., Gavrila, S., Kuhn, D. R., Chandramouli, R., Proposed NIST Standard for Role-Based Access Control, ACM Transactions on Information and Systems Security, Vol. 4, No. 3, August 2001

[FiHueOtt98] Fischer-Hübner, S., Ott, A., From a Formal Privacy Model to its Implementation, Proceedings of the 21st National Information Systems Security Conference (NISSC '98), Arlington, VA, 1998, http://www.rsbac.org/niss98.htm

[Jansen98] Jansen, W. A., A Revised Model for Role-Based Access Control, NIST IR 6192, 1998

[Ott97] Ott, A., Regelsatz-basierte Zugriffskontrolle nach dem “Generalized Framework for Access Control”-Ansatz am Beispiel Linux, Diplomarbeit, Fachbereich Informatik, Universität Hamburg, 10. November 1997, http://www.rsbac.org/dipl-ps.zip

[Ott2001] Ott, A., Rule Set Based Access Control (RSBAC), Paper for the Snow Linux Event / Unix.nl congress "Reliable Internet", Waardenburg, 14th of September 2001, http://www.rsbac.org/unix-nl

[Ott2001a] Ott, A., The Rule Set Based Access Control (RSBAC) Linux Kernel Security Extension, Paper for the 8th International Linux Kongress, Enschede, 28th to 30th of November 2001, http://www.rsbac.org/linux-kongress

[RSBAC] Ott, A., RSBAC Homepage, http://www.rsbac.org

[Sherman+95] Sherman, D. L., Sterne, D. F., Badger, L., Murphy, S. L., Walker, K. M., Haghighat, S. A., Controlling Network Communication with Domain and Type Enforcement, TIS Technical Report TISR 523, 1995