Global Information Assurance Certification Paper

Solaris RBAC Revisited

Erik Van Nooten
GSEC Practical Assignment Version 1.4b
February 12, 2003

Abstract

Role Based Access Control (RBAC) is based on two important security principles: 'separation of duties' and 'least privilege' and has the potential to reduce the complexity of security administration. With RBAC, security is managed at a level that is close to the organization's structure. Traditional security management has required the mapping of an organization's structure to a relatively low-level set of access controls, typically Access Control Lists (ACL). Although the acceptance has been slow, RBAC is attracting increasing attention. Various vendors are currently offering RBAC solutions in their products. Sun offers an RBAC facility since Solaris version 8. Although the availability of this facility helps in implementing Solaris RBAC, it is possible to use the Solaris ACL mechanism to accomplish a similar goal. At the same time, this fact is one of the major drawbacks of the Solaris implementation. The interaction of both mechanisms can make the understanding of the whole system more difficult.
Solaris RBAC does not directly support important RBAC principles such as 'role hierarchies' and 'role constraints'. It even allows implementing a non-RBAC compliant setup using RBAC features. This can increase the burden of security management by one order of magnitude higher. The objective of this document is to discuss these various Solaris RBAC scenarios and how they comply with the proposed NIST [1] RBAC standard.

Access Control Models

Various security models exist that address different aspects of security in operating systems. For example, the Bell-LaPadula model defines security in terms of mandatory access control and addresses confidentiality only, while the Biba model addresses integrity. These models are implementation-independent and provide a powerful insight into the properties of secure systems, lead to design policies and principles, and some form the basis for security evaluation criteria.
The access control model defines how users access resources ("how subjects access objects"). There are three main types access control models in use today:

- Discretionary Access Control (DAC)
  The most common way of managing user access towards resources is to assign the proper permissions to the user. Most current operating systems use an Access Control List (ACL) to accomplish this goal. ACL's are stored directly with the resources they protect. Under certain conditions, the user has the authority (=discretion) to specify what resources are accessible.

[1] National Institute of Standards and Technology

- Mandatory Access Control (MAC)
  This type of access control is based on attaching security labels to resources. These labels indicate a security classification (for example, top secret, secret, confidential and public). Users are given a specific security clearance (for example top secret, secret, confidential and public). By comparing a user security clearance (say secret) against a security label (say secret), the operating system can grant or deny access to the resource. The operating system will also check if a user with a 'secret' clearance has a need-to-know to access a document that has a 'confidential' security label. Even if the user has the appropriate clearance, the operating system can deny access to the document based on the need-to-know rules. This access control model is stricter than the others and therefore used in environments where security is of the up most importance, such as military or certain government organizations.
- Non-discretionary Access Control
  This type of access control uses a central authority to determine which users have access to what resources. Access control can be based on the role a person has within the organization (role-based) or the responsibilities and duties a person are expected to perform (task-based). It has the interesting property that the role or responsibility does not have to change when a person assumes a new role. The person is simply assigned to his new role. All permissions are assigned to the role or responsibility and not to the individual.

Solaris RBAC is an example of non-discretionary role-based access control model. RBAC itself can be found in a variety of commercial and non-commercial systems such as applications servers, web servers, database management systems and many more.

What exactly is a Role Based Access Control system?

As indicated in the name, an RBAC system uses the non-discretionary access control model and is based on role assignment and privileges or permissions associated with a particular role. The creation of roles reflects the structure of the organization.

The following principles are key concepts in the support of the Solaris RBAC system and will be used in the subsequent discussion of RBAC reference models.

Separation of Duties principle

This principle requires that two or more persons must be responsible for the completion of a task or a set of tasks. A typical example is the set "Purchasing Manager-Accounts Payable Manager". If one and the same person would carry out these roles, he would be tempted to create a fake order and approve that order. So the main purpose of 'separation of duties' is to avoid fraud, misuse and errors.
Although the principle is straight forward, the implementation is not. There are two main variations to the implementation:

Static Separation of Duties

With static Separation of Duties, two roles are strongly excluded. This means that in our example the two roles would never been assigned to the same person. It implies that this check is done during the administrative phase. Whenever a person is assigned to a new role, the system needs to verify if the new role and the already assigned roles are not mutually exclusive. Although it is the simplest variation of the two, it has the disadvantage that it does not always reflect the functioning of the organization.

Dynamic Separation of Duties

Exclusion is enforced at the session level. In its simplest form, our two roles can be assigned to the same user. The user cannot assume both roles within his set of sessions. The RBAC system must enforce that the user is logged out from the "Purchasing Manager" role when he wants to assume the "Accounts Payable Manager" role. In comparison with the previous variation, 'dynamic separation of duties' gives more flexibility to organizations.
Variations on this theme have been defined by the use of object, operational, history, order-dependent and order-independent based constraints. [1]

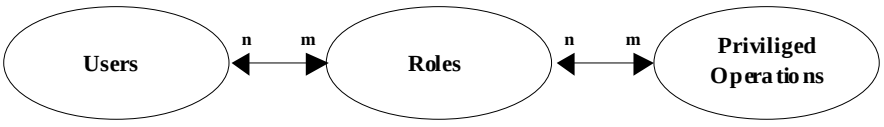
Least Privilege principle

The principle of "Least Privilege" implies that a user is given no more privileges than is strictly necessary to perform his task. There is again a dimension within this principle as it is possible to statically or dynamically assign privileges. Constraints can be placed on privileges so they do not exist longer than is required to perform a task. This is sometimes referred to as the "Timely Revocation of Trust".

RBAC Reference Models

Since it has been proven difficult to capture RBAC in one reference model, different types of RBAC reference models have been defined [2]. These models have changed over time and are currently been defined in a proposed NIST RBAC standard [3]. In a time span of 10 years, there has been significant advancement in these models. Someone who wants to study these models can become confused by the different definitions that were fashionable at a specific point in time. This is reflected in the way RBAC features in the reference models have been re-ordered over time.
The definitions from the NIST RBAC standard will be used in the remainder of the document. A summary of the differences made to the older RBAC reference models can be found in the next section.

The general RBAC model has not changed for a long time and has the following form:



The basic concept is that users are assigned to roles and that roles are associated with privileged operations. This allows users to carry out tasks with a minimum set of privileges.

Core RBAC

A set of four reference models has been defined. The Core RBAC reference model, named here **RBAC$_C$**[2], is defined as follows:

1. In the model, we define 'users', 'roles', 'privileged operations' (also dubbed 'commands') and 'sessions'.
2. Users and roles have an n-to-m relationship (n and m $\geq$ 1). One user can have different roles; one role can be assigned to different users.
3. Roles and operations have also an n-to-m relationship. One role can include different privileged operations; one operation can be assigned to different roles. Operations are being controlled by permissions.
4. Users and sessions have a 1-to-m relationship. A user can have multiple sessions going on, but a session is only assigned to one user.

These four requirements allow most group-based access control systems to comply with the NIST RBAC standard.
There is a fifth requirement included below.

5. Sessions have a one-to-one relationship with a 'Role Set Association'. A 'Role Set Association' is a subset of the roles authorized for that user.

The active Role Set Association at a particular time in a session is referred to as the 'Active Role Set' (ARS). This 'Active Role Set' may change during the lifetime of the session and can be used to enforce 'separation of duties'.

Example: Consider a real world example where a police officer is also playing basketball in his spare time. He normally carries his gun while on duty as a police officer. Carrying his gun while playing basketball will probably not be seen as very ethical. The fact that he stores his gun somewhere safe while being on duty is considered a normal procedure. There is a clear distinction between his 'Active Role' as a police officer and his 'Active Role' as a basketball player. He does not automatically have the privileges that go with one role, while he is using a different role. Enforcing only one of both roles to be active in an 'Active Role Set' is the first step towards implementing the 'separation of duties' principle.

This requirement has been relaxed as the NIST RBAC standard now explicitly states, "Core RBAC requires that users are able to simultaneously exercise permissions of multiple roles". There was a lot of debate on the differences between a role and a group in the RBAC community. [4] A user in a role was only expected to execute the privileges attached to this role. A user in a group has always the permission to execute privileges obtained from different

[2] This **bold** font type will be used when a RBAC reference model is meant. Regular capital letters will be used to talk about RBAC in general.

groups. With this change in the requirement, groups can now be used as roles. Our police officer can carry his gun while playing basketball. The fifth requirement could be interpreted as a constraint to enforce 'separation of duties'. In the NIST RBAC standard, this constraint was considered as being overly strict and has been moved to the 'Separation of Duty Relations' reference model.

Hierarchical RBAC

Hierarchical RBAC, abbreviated to $\mathbf{RBAC}_{rh}$, introduces the concept of role hierarchies. In general, hierarchies are used by organizations to deal with authority and responsibility.

Hierarchies can be used to inherit permissions from a previous role. However, there is a difference between role permissions inheritance and permissions of two roles active at the same time in the same session. In the first case, a new role is created and inherits the privileges of the first role. In the second case, an 'Active Role Set' cannot be enforced. The latter case boils down to the discussion on the 'Active Role Set' and the fact that users are allowed to exercise their permissions simultaneously as discussed in the previous section.

The NIST RBAC standard recognizes two types of role hierarchies:

- General Hierarchical RBAC
  In this case, a role can inherit permissions from multiple different roles.
- Limited Hierarchical RBAC
  In this case, a role can inherit permissions from only one immediate descendant.

Static Separation of Duty Relations (SSD)

The next two RBAC reference models introduce the concept of constraints. Typically, conflict of interests is avoided when roles are *mutually excluded*.

Example: The following are typical cases of such roles: "System Administrator-User Security Management" and "Purchasing Manager - Accounts Payable Manager". A 'mutually exclusive' constraint will enforce 'separation of duties'.

Constraints can also play a role in the way $\mathbf{RBAC}_{rh}$ behaves. *Inheritance* of privileged operations can be blocked by a constraint, for example.

This is the reason why two types have been defined:

- Static Separation of Duty
  This is the classic case where a user may be prohibited to be assigned to a role because of the character of already assigned roles. This property is enforced in an administrative way.
- Static Separation of Duty in the Presence of a Hierarchy
  This type of relation works in the same way, except that it applies to inherited roles as well as directly assigned roles.

Dynamic Separation of Duty Relations (DSD)

As can be deducted from the name, constraints are being enforced in a dynamic matter. A constraint can be placed on a user session and as a result, a user can only use one *single session*. A constraint could enforce all open sessions to switch to the active role ('*role enforcement*') at the moment the user assumes his role. DSD extends the support for the least privilege principle in the sense that each user needs different permission levels at different times. DSD makes sure that permissions do not persist beyond the time that they are required. "Timely Revocation of Trust" is the attained goal for DSD.

NIST RBAC standard and prior Reference RBAC models

Since some RBAC papers reference earlier RBAC reference models, it is useful to point out the differences of these models with the proposed NIST RBAC standard.

Core RBAC is equivalent to what was called "Minimal RBAC". [4] "Minimal RBAC" was a relaxation of $RBAC_0$, which included the fifth requirement as discussed in the Core RBAC section above. Core RBAC is sometimes referred to as 'flat RBAC'.

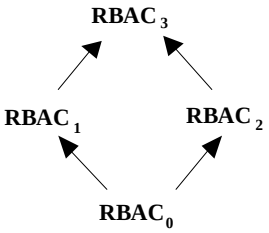
In $RBAC_0$, a user could not exercise all his assigned role permissions at all times. The enforcement of an 'Active Role Set' was part of the default requirements.

$RBAC_1$ is equivalent with hierarchical RBAC, although there was no difference between 'general' or 'limited' hierarchical RBAC.

Both the SSD and DSD models were combined in one model, named $RBAC_2$. This model addressed all constraints.

$RBAC_3$ was the highest form of RBAC and was the consolidation of $RBAC_0$, $RBAC_1$ and $RBAC_2$.

The relation between these Role Based Access Control Reference Models is as follows:



$RBAC_3$ has some issues with multiple inheritances supported in the model. The split of role hierarchies and constraints can lead to inconsistencies. Suppose two roles are defined as being 'mutually exclusive'. A new role can be defined via role hierarchies, which inherits these two 'mutually exclusive' roles.

	$RBAC_c$ (Core or Flat RBAC)	$RBAC_{rh}$	$RBAC_{ssd}$	$RBAC_{dsd}$
Minimal RBAC	√	-	-	-
$RBAC_0$	√ *minus execution of all role permissions*	-	-	-
$RBAC_1$	-	√	-	-
$RBAC_2$	-	-	√	√
$RBAC_3$	-	*Distributed over the different models*		

The objective of the NIST RBAC standard was to include all group-based access control mechanisms, which were excluded in the earlier RBAC definitions. A formal functional specification can be found in appendix 1 of the NIST RBAC standard. [3]

RBAC within Solaris

Solaris leaves the administrator the choice to implement RBAC using Access Control Lists (ACL) or the Solaris RBAC facility.
When ACL's are used, groups are configured as roles. A user is a member of a group and therefore also member of a role. This means that a user will always be able to exercise all his privileges all the time as is allowed within Core RBAC.
In most operating systems [3], permissions can be set at the user or the group level. In order to have an ACL system function as a RBAC system, only groups must be used as entries in the ACL. Most systems, including Solaris, have no option to enforce this.

Classical Unix

The classical Unix permission system is limited in its functionality. There are only three groups on which one can place permissions: the **u**ser, the **g**roup to which the file belongs and everyone else (**o**ther). An example is shown below:

```
$ touch foo
$ ls -l foo
-rw-r--r--   1 Erik     sysadmin       0 Dec 29 09:06 foo
```

Let's set up a simple role model by introducing Victor and Elizabeth. Victor is a security operator and Elizabeth works as security officer. Victor and Elizabeth belong to different groups: Victor is member of the 'secops' group and Elizabeth is member of the 'secoffs' group. Both are members of the general 'staff' group. The 'secops' group has different permissions than the 'secoffs' group. The 'secops' group could be authorized to add, modify and delete users, while the 'secoffs' group can change passwords [4].
It is possible to enforce 'separation of duties' by creating the proper scripts or programmes. In this example, the security officers will use the script *foo1*, while the security operators will use *foo2* as shown below:

[3] Sun Solaris, MS Windows NT, MS Windows 2000…

[4] This is just an example to show how different roles can be created. It is not meant to serve as a full-featured security role model.

```
$ ls -l
total 6
---s--x---   1 root     secoffs      115 Dec  29 09:29 foo1
---s--x---   1 root     secops       150 Dec 29 09:34 foo2
```

As can be seen from the above output, the 'setuid' bit has been set on the files ('s' in fourth position of the permissions' set). The 'setuid' bit allows the script to run with an effective userid of 0. So Victor would not be able to execute the security officers' scripts, while Elizabeth will not be able to execute security operators' scripts.

```
$ su Victor
Password:
$ ./foo1
ksh: ./foo1: cannot execute
$ ./foo2
This is foo2 -> a file that is common to security operators!
uid:Victor(103) euid:root(0) gid(s):staff(10) secstaff(501) secops(503) egid:staff(10)
$ su Elizabeth
Password:
$ ./foo1
This is foo1 -> a file that is common to security officers!
uid:Elizabeth(104) euid:root(0)  gid(s):staff(10) secstaff(501) secoffs(502) egid:staff(10)
$ ./foo2
ksh: ./foo2: cannot execute
```

The script above is executed with an effective uid 0 ('euid:root(0)'). Some commands require that the real uid must be root as well [5]. There is a difference between the Bourne shell (/usr/bin/sh) and the Korn shell (/usr/bin/ksh) on setting the uid. The Bourne shell will always make the uid the same as the euid unless the ' -p ' option is specified. The Korn shell has the ' -p' option specified as default and will use the /etc/suid_profile when the effective uid is not the same as the real uid.

The implementation of such a model has limited flexibility and some serious drawbacks:

1. If there is a need to have common scripts that need to be executed by both groups, all users of both groups will have to be members of a third group ('secstaff' in the example below).
2. An administrator has to set-up and maintains various scripts (foo, foo1 en foo2), which includes the various commands that should run with effective uid of 0.
3. An 'Active Role Set' cannot be enforced. Both Victor and Elizabeth will always have the permission to execute their scripts.
4. Permissions are stored with the resource as is the case by setting the execute permission. The 'setuid' bit can be placed on any file in the system, although they are generally grouped at one location. This means that a system administrator has to query the entire system to have a view on who has what privilege.

[5] The /usr/bin/passwd is such a command.

```
$ id -a
uid=103(Victor) gid=10(staff) groups=10(staff),501(secstaf    f),503(secops)
$ ls -l foo
---s--x---   1 root     secstaff     133 Dec 29 09:25 foo
$ ./foo
This is foo  -> a file that is common to both security operators and officers!
uid:Victor(103) euid:root(0) gid(s):staff(10) secstaff(501) secops(503) egid:staff(1      0)
$ su Elizabeth
Password:
$ id -a
uid=104(Elizabeth) gid=10(staff) groups=10(staff),501(secstaff),502(secoffs)
$ ./foo
This is foo  -> a file that is common to both security operators and officers!
uid:Elizabeth(104) euid:root(0) gid(s):staff(10) secsta    ff(501) secoffs(502) egid:staff(10)
```

File Access Control List

A new ACL mechanism, named File Access Control List (FACL), was introduced from Solaris 2.5 onwards. The implementation is POSIX 1003.6 compliant. Two operating system commands ("getfacl" and "se tfacl") extend the classical ACL system. [5] It allows more flexibility than the traditional permission bits on a file or directory.

The advantage of using the 'facl' mechanism is that we can get rid of the 'secstaff' group in the previous section (first b ullet in the above-mentioned drawback list). Permissions can be assigned per group.

```
$ getfacl foo*

# file: foo
# owner: root
# group: other
user::--x
group::---          #effective:--
group:secoffs:--x          #effective:--x                ⇐ assign permission to both
group:secops:--x           #effective:--x                ⇐ groups
mask:--x
other:---

# file: foo1
# owner: root
# group: other
user::--x
group::---          #effective:--
group:secoffs:--x          #effective:--x                Ü only secoffs can execute
foo1
mask:r-x
other:---

# file: foo2
# owner: root
# group: other
user::--x
group::---          #effective:--
group:secops:--x           #effective:--x                Ü only 'secops' can execute
foo2
mask:r-x
other:---
```

```
$ id -a
uid=103(Victor) gid=10(staff)   groups=10(staff),503(secops)
$ /foo
This is foo  -> a file that is common to security operators and officers!
uid: Victor(103) euid:root(0) gid(s):staff(10) secops(503) egid:staff(10)
$ /foo1
ksh: ./foo1: cannot execute
$ /foo2
This is foo2  -> a file that is common to security operators!
uid: Victor(103) euid:root(0) gid(s):staff(10) secops(503) egid:staff(10)
```

Some observations

- The group owner of the file does not have to be related to the target executable group.
- The 'facl' mask field holds the maximum value that can be achieved. In the above files, the mask field equals 'r -w' and the group field equals '—x'. This leads to an effective field of ' —x'.
- When a 'facl' permission is being set, there will be a '+' next to the permission bits.

```
$ ls -l
total 6
---s------+  1 root     other      128 Dec 29 09:27 foo
---s------+  1 root     other      115 Dec 29 09:29 foo1
---s------+  1 root     other      116 Dec 29 09:29 foo2
```

- File Access Control Lists are set with the following commands:

```
# setfacl -s u::--x,g::--,g:secoffs:--x,g:secops:--x,m:--x,o:---  foo
# setfacl -s u::--x,g::--,g:secoffs:--x,m:r-x,o:---  foo1
# setfacl -s u::--x,g::--,g:secops:--x,m:r-x,o:---  foo2
```

RBAC using the Solaris RBAC model

The RBAC model on Solaris is an implementation of the $RBAC_c$ reference model. The main attributes of the Solaris RBAC model are:

1. Although it <u>does</u> support an 'Active Role Set', only one role can be placed in the 'Active Role Set'. The set cannot be changed dynamically.
2. It <u>does not</u> support role hierarchies ($RBAC_{rh}$).
3. It <u>does not</u> support $RBAC_{ssd}$ or $RBAC_{dsd}$. The security administrator must make sure that two mutually exclusive roles are not assigned to the same user.

The module responsible for this native support is named 'pam_roles' and is located in /usr/lib/security/$ISA directory. The module can be activated or deactivated by configuring the appropriate entry in the /etc/pam.conf directory. It is possible to replace this module by a customized version. A sample version is available from the Sun web site. [6]
During a normal installation, the module is activated and the /etc/pam.conf directory contains the following entries:

```
$ grep pam_roles /etc/pam.conf
login   account requisite     /usr/lib/security/$ISA/pam_roles.so.1
dtlogin account requisite     /usr/lib/security/$ISA/pam_roles.so.1
other   account requisite     /usr/lib/security/$ISA/pam_roles.so.1
ppp     account requisite     /usr/lib/security/$ISA/pam_roles.so.1
```

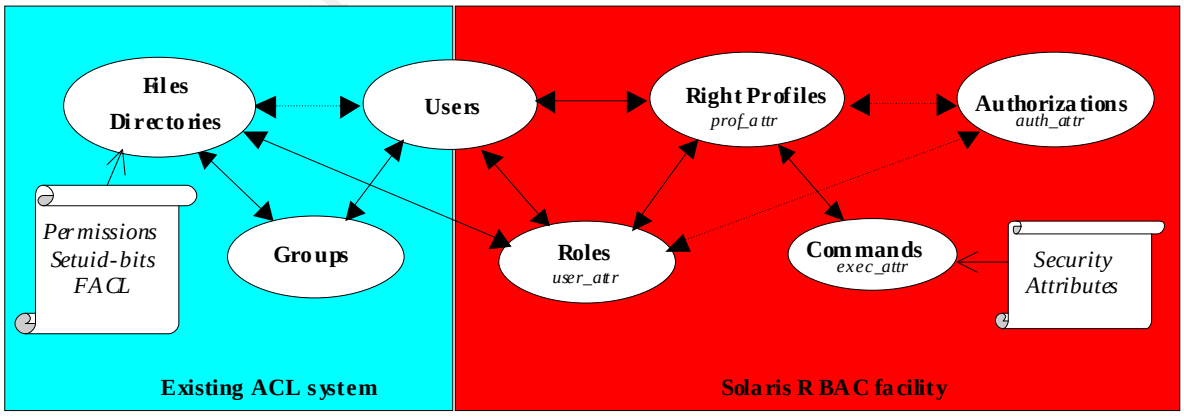
A complete documentation on how to set-up the Solaris RBAC facility is found in [7]. We will only discuss the main configuration files.

In order to manage a role, three commands are used: *roleadd*, *roledel* and *rolemod*. These commands manipulate the following four system files:

1. /etc/user_attr — associates roles and profiles to users
2. /etc/security/auth_attr — container for authorizations
3. /etc/security/prof_attr — container for right profiles
4. /etc/security/exec_attr — container for execution profiles

A particularity of the Solaris RBAC facility is that a role is created as a regular userid. However, it is not possible to login directly using this role-userid. The *su* command has to be used to change to the role.
Solaris RBAC extends the permission model by introducing "right profiles", "commands and their security attributes" and "roles". All RBAC privileges can be managed in a centralized way (last bullet in the above-mentioned drawback list).

The different assignment possibilities of both the ACL and Solaris RBAC facility are depicted in the diagram below. As can be seen, right profiles, authorizations and commands can be assigned to roles or directly to users. ACL's can be assigned to users, groups and roles. The multitude of possible assignment relations is the primary reason that an existing RBAC set-up can be confusing at times.



In order to continue our example, the following right profiles [6] have been created:

```
$ grep SEC /etc/security/*attr*
/etc/security/exec_attr: SECOFF:suser:cmd::/usr/bin/passwd:uid=0
/etc/security/exec_attr: SECOFF:suser:cmd::/export/home/Erik/giac/secoff/foo:euid=0
/etc/security/exec_attr: SECOFF:suser:cmd::/export/home/Erik/giac/secoff/foo1:euid=0
/etc/security/exec_attr: SECOP:suser:cmd::/usr/sbin/useradd:euid=0
/etc/security/exec_attr: SECOP:suser:cmd::/usr/sbin/usermod:euid=0
/etc/security/exec_attr: SECOP:suser:cmd::/usr/sbin/userdel:euid=0
/etc/security/exec_attr: SECOP:suser:cmd::/export/home/Erik/giac/secoff/foo:euid=0
/etc/security/exec_attr: SECOP:suser:cmd::/export/home/Erik/giac/secoff/foo2:euid=0
/etc/security/prof_attr: SECOP:::Security Operator::help=SecOp.html
/etc/security/prof_attr: SECOFF:::Security Officer::help=SecOff.html
```

Assigning right profiles to users directly

This option is advised against in the Sun documentation [7]: "Right profiles and authorizations can also be assigned directly to users. This practice is discouraged because it enables users to make mistakes through inadvertent use of their privileges". Although this is correct, this practice is allowed in the $RBAC_c$ reference model.

A bigger problem is that this option effectively bypasses the creation of roles. The role is associated with the group setting as is being described in the $RBAC_c$ model. Permissions are active at all times. So, it looks like we can implement an effective RBAC model by 'augmenting' the ACL system with privileges assigned via right profiles or authorizations. This seems to create a model that is compliant with the $RBAC_c$ reference model. By the way, this is a case where the two methods to implement RBAC on a Solaris system are mixed. The 'roles' are in fact borrowed from the ACL-based group access control, while the permissions are added from the Solaris RBAC facility by assigning right privileges or authorizations.

However this set-up has a major limitation. It violates one of the RBAC principles since privileges are assigned directly to the users and **not** to the roles (hence groups).

Example: Suppose we have 4 users to whom we want to assign 1 right profile and 1 authorization. If we use a role to assign the permissions, we have 6 relations: 4 users connected to one role, one role connected to 2 privileges. For N users and M permissions, there is an N+M relation. In the event we assign the permissions directly to the users, we will have 8 relations. Each user will have 2 privileges. So for N users and M permissions, there is an NxM relation. This is an order of magnitude higher ($O(N^2)$ versus $O(N)$) than the previous case. If there are a lot of users and permissions, these relations can become quickly unmanageable.

After assigning the privileges to our user, we can invoke the following session:

```
$ su Victor
```

[6] We only assign "commands with security attributes" in this example. Solaris has also the concept of authorizations, which allows GUI applications to check for permissions in an equivalent manner. A standard set of authorizations ('solaris.*') is defined within the Solaris system.

[7] [6] Page 247

```
Password:
$ profiles -l

    SECOP:
      /usr/sbin/useradd    euid=0
      /usr/sbin/usermod    euid=0
      /usr/sbin/userdel    euid=0
      /export/home/Erik/giac/secoff/foo    euid=0
      /export/home/Erik/giac/secoff/foo2    euid=0
    All:
      *
```

The 'profiles' command shows security attributes that have been set for files 'foo' and 'foo2'. The 'euid=0' keyword is equivalent to setting the 'setuid' -bit in the previous examples. It is possible to use the 'uid=0' keyword in order to have a real uid of 0. It is also possible to specify different uid values.
We can assign specific commands that need root permissions to a specific profile (second bullet in our above-mentioned drawback list).

```
$ roles
roles: Victor : No roles
```

Since the profile has been assigned directly to the user, no roles are assigned to the user. The user Victor directly has the permission all the time.

```
$ ls -l
total 6
---x------   1 root     other        128 Dec 29 09:28 foo
---x------   1 root     other        116 Dec 29 09:30 foo1
---x------   1 root     other        116 Dec 29 09:30 foo2
$ `pwd`/foo
This is foo -> a file that is common to security operators and officers!
uid:Victor(103) euid:root(0) gid(s):staff(10) secops(503) egid:staff(10)
$ `pwd`/foo1
pfksh: /export/home/Erik/giac/secoff/foo1: cannot execute
```

Note that the 'pfksh' is used. The 'profile Korn' shell is similar to the Korn shell with the additional characteristic that it can understand the different authorization requests needed to support the Solaris RBAC facility. Similar shells exist for each of the traditional shells. The names of the Bourne and C counterparts are 'pfsh' and 'pfcsh'. All shells can execute the 'pfexec' command. This program takes arguments from the shell and executes them with specified security attributes obtained from the execution profile.

```
$ `pwd`/foo2
This is foo2 -> a file that is common to security operators!
uid:Victor(103) euid:root(0) gid(s):staff(10) secops(503) egid:staff(10)

$ id -a
uid=103(Victor) gid=10(staff) groups=10(staff)
$ useradd joe                                    ⇐ adding user 'joe'
$ usermod -s /usr/bin/ksh joe                    ⇐ change his default shell
$ grep joe /etc/passwd
joe:x:105:1::/home/joe:/usr/bin/ksh
$ passwd joe                                     ⇐ trying to change password
passwd (SYSTEM): Permission denied
```

```
passwd (SYSTEM): Can't change local passwd file

Permission denied

$ su Elizabeth
Password:
$ profiles -l

    SECOFF:
      /usr/bin/passwd    uid=0                                  ⇐ passwd cmd in privileges
      /export/home/Erik/giac/secoff/foo    euid=0
      /export/home/Erik/giac/secoff/foo1    euid=0
    All:
      *
$ roles
roles: Elizabeth : No roles
$ `pwd`/foo
This is foo -> a file that is common to security operators and officers!
uid:Elizabeth(104) euid:root(0) gid(s):staff(10) secoffs(502) egid:staff(10)
$ `pwd`/foo1
This is foo1 -> a file that is common to security operators!
uid:Elizabeth(104) euid:root(0) gid( s):staff(10) secoffs(502) egid:staff(10)
$ `pwd`/foo2
pfksh: /export/home/Erik/giac/secoff/foo2: cannot execute

$ passwd joe                                                   ⇐ changing Joe's password
New password:
Re-enter new password:
passwd (SYSTEM): passwd successfully changed for joe
```

Although this set-up is frequently used as an RBAC example in literature [8][9], it violates one of the RBAC principles due to the direct assignment of user privileges. The management advantage of assigning permissions to roles is lost. The set-up is only viable if a small number of users would receive a small number of direct assigned privileges. This can be considered acceptable for the assignment of system userid's that are used in background jobs, i.e. daemon userid's that require certain privileges. However, if the number of these userid's and/or privileges becomes large, the management grows with an order of magnitude compared to the next set-up.

Assigning roles to users

In order to assign roles to users and implement an effective $RBAC_c$ model, two roles are being created: 'rr_secop' and 'rr_secoff'.

```
$ su Victor
Password:
$ roles
rr_secop
$ profiles -l

    All:
      *
$ su rr_secop                                                  ⇐ need to change to role
Password:
$ profiles -l
```

```
    SECOP:                                                    ⇐ more privileges added
      /usr/sbin/useradd    euid=0
      /usr/sbin/usermod    euid=0
      /usr/sbin/userdel    euid=0
      /export/home/Erik/giac/secoff/foo    euid=0
      /export/home/Erik/giac/secoff/foo2    euid=0
    All:
      *
$ `pwd`/foo
This is foo  -> a file that is common to security operators and officers!
uid:rr_secop(102) euid:root(0) gid(s):secops(503) egid:secops(503)
$ `pwd`/foo1
pfksh: /export/home/Erik/giac/secoff/foo1: cannot execute
$ `pwd`/foo2
This is foo2  -> a file that is common to  security operators!
uid:rr_secop(102) euid:root(0) gid(s):secops(503) egid:secops(503)
```

Using this model, we can define an **RBAC$_c$** reference model and enforce an Active Role Set (third bullet in the above-mentioned drawback list). One important thing to note is that all permissions (ACL's) have to be set to the role and that there is **no** inheritance of privileges. In the previous examples, all permissions were active.

The Solaris RBAC facility is limited in regard to the NIST RBAC standard **RBAC$_c$** reference model as it only allows one role to be in the 'Active Role Set'. Multiple roles per user can be activated, but they need to be separated in different sessions.

Auditing

When auditing is active via the Basic Security Module (BSM) module, we can track who executed the different commands as shown below. Changing roles does not change the 'audit id'.

```
# ps -edf|grep secop
   root  1342   798  0 22:08:19 pts/3    0:00 grep secop
rr_secop  1337  1335  0 22:07:47 pts/4    0:00 pfksh
# auditconfig -getpinfo 1337
audit id = Erik(100)
process preselection mask = ex,lo(0x40001000,0x40001000)
terminal id (maj,min,host) = 0,0,sundance(172.31.201.7)
audit session id = 313
```

Conclusion

Solaris makes it possible to build various RBAC implementations. Having a Solaris RBAC facility on one hand and the possibility to implement a RBAC system using Discrete Access Control Lists (DACL) on the other hand, can be confusing in understanding what part of the ACL and/or RBAC facility is being used.

The choice of building an RBAC model using one of both models is entirely left open as an implementation choice.

The assignment of RBAC right privileges directly to users is not considered as being a good practice. It violates one of the RBAC principles that says that permissions should be assigned to roles or groups. The only reasonable use for this set-up is the assignment of right privileges to daemon userid's requiring certain privileges. Daemon userid's normally come in small sets.

Since the number of relations is in this case an order of magnitude higher, security may become unmanageable for large sets of userid's and privileges.
Solaris RBAC has some major limitations, as it does not include the possibility to use role hierarchies and role constraints. RBAC reference models such as 'General and Limited Hierarchical RBAC and 'Static and Dynamic Separation of Duty can only be supported by heavily extending the 'pam_roles' module in the RBAC facility. Solaris RBAC only allows one role from the 'Active Role Set' to be active in the same session.
Convergence is on the way between the various RBAC reference models as the National Institute of Standards and Technology (NIST) has proposed the first RBAC standard in 2001. Hopefully, vendors will endorse the NIST RBAC standard and release compliant products in the near future. This will limit the actions that users of a computer system can perform and help security administrators to have more comprehensive view on the distribution of privileges.

References

1. M. E. Zurko & R. T. Simon. "Separation of Duty in Role-Based Environments". 1997
URL:http://citeseer.nj.nec.com/cache/papers/cs/22050/http:zSzzSzwww.memesoft.comzSzadagezSzsep-duty.pdf/simon97separation.pdf

2. Ravi S. Sandhu et al. "Role Based Access Control Model". October 26, 1995
URL:http://citeseer.nj.nec.com/cache/papers/cs/872/http:zSzzSzwww.list.gmu.eduzSzjournalszSzcomputerzSzps_verzSzi94rbac.pdf/sandhu96rolebased.pdf

3. National Institute of Standards and Technology (NIST). Proposed NIST RBAC standard. August 2001
URL:http://csrc.nist.gov/rbac/rbacSTD-ACM.pdf

4. John Barkley. "Comparing Simple Role Based Access Control Models and Access Control Lists". August 11, 1997
URL:http://citeseer.nj.nec.com/cache/papers/cs/20857/http:zSzzSzwww.itl.nist.govzSzdiv897zSzstaffzSzbarkleyzSziirf.pdf/barkley97comparing.pdf

5. Sun Solaris 8 Product Documentation. "setfacl" man pages. July 23, 1998
URL:http://docs.sun.com/db/doc/806-0624/6j9vek5ge?q=getfacl&a=view

6. Solaris PAM documentation. "Pluggable Authentication Modules".
URL:http://wwws.sun.com/software/solaris/pam/

7. Sun Solaris 9 Product Documentation. "System Administration Guide – Security Services". May 2002
PartNo 806-4078-10 - Chapters 18, 19 and 20
URL:http://docs.sun.com/db/doc/816-4883?q=security+services

8. Sun Solve FAQ ID 3280. "Example of Role Based Access Control (RBAC)
URL: http://sunsolve.sun.com/private-cgi/retrieve.pl?doc=faqs%2F3280&zone_32=RBAC January 03, 2001

9. Occhinpinit, Christine. "RBAC in the Real World". September 16, 2002
URL: http://rr.sans.org/casestudies/RBAC.php

10. Maurice J. Bach. "The design of the Unix Operating System". 1986
Prentice Hall: ISBN 0-13-201757-1