The following paper was originally presented at the
Seventh System Administration Conference (LISA '93)
Monterey, California, November, 1993

Role-based System Administration or Who, What, Where, and How

Dinah McNutt
Tivoli Systems

For more information about USENIX Association contact:

1. Phone: 510 528-8649
2. FAX: 510 548-5738
3. Email: office@usenix.org
4. WWW URL: http://www.usenix.org

Role-based System Administration or Who, What, Where, and How

Dinah McNutt – Tivoli Systems

ABSTRACT

Traditionally, access for performing system administration tasks is an all or nothing proposition. With root access, an administrator can potentially make many changes to a system even though you may only want to allow them to add a user or mount a filesystem. In addition to specific tasks, you may want to control what tasks an administrator can perform based on which machine they are using. For some tasks, you also want to manage how those tasks are performed. For instance, when you add a user, you usually want to make sure the user ID is unique and is not zero.

This paper defines requirements for a role-based system administration environment. It describes and compares traditional solutions such as restricted shells, multiple root accounts, and setuid programs. The comparisons are made in the context of the requirements defined and are used to introduce the motivation and need for an alternative solution.

The solution proposed in this paper is object oriented and is based on the draft POSIX 1003.7 standard. Where appropriate, specific implementations (such as the Tivoli Management Environment) will be referenced. These examples will include lessons learned at Tivoli in developing and using an object-oriented system administration tool.

Introduction

UNIX treats almost all resources as files: disk drives, terminals, and executables. A standard file access mechanism is used to determine which process has what type of access to other resources (e.g., files, other processes, devices, etc.) This mechanism uses owner, group, and world permissions to allow read, write or execute access. On some UNIX systems, access control lists are supported which allow you to set exceptions to the standard file access for individual users or groups of users.

Most system administration functions such as adding new users, adding new devices, and editing system files, must be done as the root user. If a user can become the root user (by either using the su command or logging directly onto the system as root), the user can by-pass the traditional UNIX file protections. This is a security problem because the user can now have access to files they wouldn't normally be able to access. Traditionally, this problem has been treated as an ethical one and good system administrators do not go around and reading files that belong to other users unless it is necessary to troubleshoot a problem. As UNIX gains popularity in commercial environments, data security requirements may dictate that good ethics are not enough to solve the problem.

In a production environment, logistical concerns loom even larger. Why should I have to give someone the root password just so they can create users or mount a filesystem? Let's look at some of the issues and define some requirements for system administration tools to help solve these problems.

Requirements

The requirement for a system administration environment are summarized by the following:

- *Accountability* - Who is logged on as root? If everyone uses the root account to do system administration tasks, it is difficult to know which administrator is logged onto the system.
- *Auditing* - Who did what and when? This requirement supplements accountability by telling you what the administrators are doing.
- *Defining task-based roles* - Ability to allow different administrators to do different tasks. I may want "Joe" to be able to add users and do system backups, but I want "Mary" to be able to reboot the system and add users. This is a mechanism above and beyond the all or nothing capabilities of the root account.
- *Automation* - Automating redundant tasks should be simple. You can write scripts or program to automate tasks, but sometimes writing the tools is more effort than doing the task. A good system administration tool should make this task easy.
- *Distributed solution* - Don't repeat the same task on each of your 100+ systems. You should be able to execute one command (or series of commands) and have changes take effect on all systems.
- *Extensibility* - You should be able to customize vendor software if needed. Vendors

should supply as complete a solution as possible, but every site is different and you may have some exceptions the vendor did not anticipate.

- *Heterogeneous* - Most sites are not homogeneous and it would be ideal to have a system administration tool that ran on every system you have. Operations and commands should work transparently across systems.

Traditional Solutions

Multiple root accounts

Some sites control who can log in as root by creating multiple accounts with a UID of 0. Here is an example of multiple accounts using a UID of 0:

```
kernie:T1xig45qKWShc:0:10:kernie:
     /home/kernie:/bin/csh
kiva:K1EW6bcwJan7s:0:10:kiva:
     /home/kiva:/bin/csh
otto:21WByNSB8jMPY:0:10:otto:
     /home/otto:/bin/csh
```

Each of these users probably has a different account they use for normal activities and these special accounts used for tasks that require root access. Since each of these root accounts is independent, once a user no longer needs the root access, the special root account can be removed.

The most obvious problem with this approach is that most UNIX applications use the account UID instead of the login name. Once a user logs in using a "second root" account, the user in indistinguishable from the "real root". For example:

```
rockytop% su kernie
Password:
rockytop# whoami
root
```

What has happened is that the (*whoami* command has taken the UID 0, looked it up in the password file, and returned the first entry in the password file with a UID of 0. This is also true when you edit files. The user's UID is stored in the inode of the file, not the username.

Multiple root accounts do allow you to assign private root passwords, so you can give out selected root access to machines, without disclosing the root password used everywhere (if you use NIS or NIS+ for distributing the root password). However, this approach provides little information about who is doing what, since all actions appear to have been performed by the "root" user.

Restricted Shells

Restricted shells allow you to define a restricted environment from which users can run a limited number of commands. The traditional interactive shells (*sh*, *csh*, *ksh*, etc.) allow the user to run potentially any command. A restricted shell allows you to define which users can run what commands.

There are several restricted shells available: *sudo*, *resh*, *rsh*, and *sush* are examples. In general, these shells share the following features:

- The ability to define which commands a user may execute on a per user basis.
- Logging of who executed what command.
- Password protection so that each user must enter their password when accessing the restricted shell.

Some shells are interactive where you type the name of the shell, then execute commands just like you would from the C-shell or Bourne shell. Other restricted shells require that you pre-fix the command you wish to execute with the name of the shell. For instance:

```
sudo /bin/tar cvf /dev/rst0
       /usr/local
```

You need to be careful to not give the users access to commands that could be used to by-pass the restricted environment (like *wftp* and *wvi*.) There is a special version of *wvi* called *wrvi* available that does not allow you to escape to a shell.

The advantages of restricted shells are the flexibility and accountability they provide. One disadvantage is that you must install and maintain the source code for the shell since a restricted shell is not standard on all UNIX systems. The exception is that *wrsh* is available on System V. Many large companies would prefer to pay a vendor to provide this functionality and free their staff for other tasks. Another disadvantage is that you must configure the users of the shell on each system you wish to manage. This could be a very time consuming process if each system has different administrators.

Captive Accounts

The biggest different between a restricted shell and a captive account is that a user can start a restricted shell from his or her own interactive account, but must log in to a captive account directly in order to gain access to the environment defined by the captive account.

Often, these shells must be executed by the root user. To use the shell, simply create a special account for the restricted shell user and give the account UID 0 as if you were creating a second root account, However, define the interactive shell in the password file to be a restricted shell, and you can limit what each root-privileged user can do. The same advantages and disadvantages for restricted shells apply for captive accounts.

Setuid programs

Setuid programs are set to mode 4755 or something similar. The setuid bit, 04000, is the important part. Users who execute these programs actually run as owner of the file for the duration of the program.

If you have a very specific task, you can write a C program or perl script that contains all the commands needed to do the task. Set the owner of the file to be root and the file mode 4755. Then any user can execute the file. You can also set the mode to 4750 and the group ownership to a specific group (e.g., *rootusrs*.) Then you can add any users you want to be able to run this script to the *rootusrs* group.

Setuid programs are useful to delegate specific tasks to a set of users as long as you have a manageable set of tasks. A complex site with many combinations of systems and pseudo-privileged users may not find this a practical solution for many tasks. In addition, be cautious with setuid scripts, since they are a common source of security holes.

An Object Oriented Solution

The basic problem we are trying to solve is "Add Joe User to these 50 systems."The fact that you have 10 HP/UX systems, 10 Suns, 20 AIX systems, 9 SGIs, and 1 NeXT is irrelevant. It would be nice to execute the *add_user* command and have all the right things happen to each system.

As a system administrator, you want to know what the *add_user* command is doing and be able to modify the procedures it calls to add this user. Once it is set up, you should to be able to run the command without worrying about what it is doing.

Under this scenario, as more types of systems are added to the network, you would only have to make minor changes to the *add_user* procedure on the new host.You wouldn't have to train your staff how to administer the system since they would already know how to use the system-independent system administration commands.

This solution has three basic components: a command-line interface, a programmatic interface, and a set of managed objects. Some specific applications may also include a GUI-based interface. The managed objects encapsulate the information required to administer the objects. In other words, the user object on an AIX system would contain the commands required to add a user to that system. The fact that these commands are different than those required to add a user to a Solaris system would be transparent.

This strategy is based on the POSIX 1003.7 standard and assumes that a management platform or framework that supports communication between the different systems and supports the implementations of a common set of managed objects (users, groups, printers, etc.) exists on all systems. This framework includes a common API to the framework which would provide access to the services provided by the platform, including the ability to access objects whose locations are unknown to the application. This framework is outside the scope of POSIX 1003.7 as it is being addressed by OSF, UI, ISO/OSI, COSE and others.

Managed Objects

The POSIX 1003.7 (or POSIX.7) standard attempts to define managed-object classes that are descriptions of both the properties (attributes) and behavior (methods) of logical and physical resources that are managed on a system or systems. Examples of managed-object classes include user, printer, and host. Properties of the user class might be login name or home directory. Behavior methods would perform tasks like creating the home directory or changing the user's password. This reference model

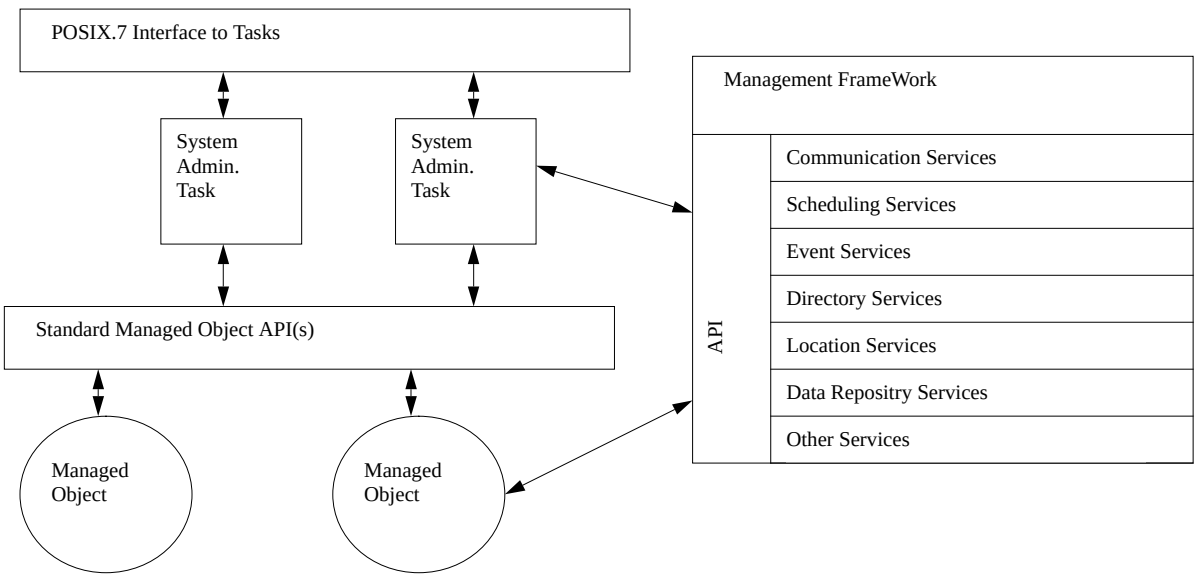

Figure 1: POSIX.7 Reference Model

is shown in Figure 1. Assumptions made by this model include:

- System administration is a task-oriented process.
- A task is a high-level operation, such as "export filesystem" or "delete user".
- There are three different interfaces: a command line interface (CLI), and applications programming interface (API), and a graphical user interface (GUI.) POSIX.7 only addresses the CLI and API.
- A task may internally invoke other tasks.
- The management framework provides common services such as naming/location, notification, data management, communication, authentication, and authorization.
- The location of the managed objects is transparent.
- Management operations are typically simple, policy-free, low-level atomic operations (such as changing a user's UID in the password file or disabling FTP service on a host.)

In this environment, applications work as sets of cooperating objects. These object must delegate authority to other objects in order for tasks to execute correctly. The delegation must happen in a secure manner to make sure someone is not able to perform a task they have not been authorized to perform. Role-based security is one way to implement a secure delegation mechanism.

Roles

As mentioned earlier, there is a need to perform system administration tasks without using the all or nothing capabilities of the root account. The management environment described in the previous section provides a mechanism for defining different roles for different administrators. Since all the information needed to manipulate an object is encapsulated within the object, you can extend the attributes of the object to include roles. A simple role-based model could be patterned after UNIX file system access: read/write,read-only, or none. If you have "read/write" access to an object, you can modify or delete the object by invoking the one or more of the object's behavior methods. "Read-only" access allows you to examine the attributes of an object and "none" would not allow you to view any information about the object.

The next step is to identify the system administrators to the management environment. Ideally, I want to be able to log onto a system as "dinah" and perform tasks under that user ID. The management system must be able to provide some kind of authentication that I really am "dinah" and determine what kind of authorization I have. This implies that authorization and roles must be pre-defined by one of the administrators. Let's look at the following matrix:

	Printers	Users	File Systems
Jamie	RW	RW	N
Larry	RO	N	RW
Shoe	N	RW	N

Using the Read/Write, Read-only, and None notation described earlier, you can define access for three different administrators with distinct, but overlapping job functions. If Jamie executes the *add_user* command, the management environment would determine what access was required to add a user by getting the information from the attribute on the user object. Next, it would look up what access Jamie has been authorized with respect to that task. The two must match before the task will be performed.

In reality, you want more complex operations than read/write, read-only, and none. By carrying this model one step further, we can define specific administration tasks. For instance, let's define the task "Manage Printers". This high-level task might include the following steps:

- Reset print queue
- Define new printer
- View print queue
- Delete job from queue
- Re-order print queue
- Delete printer
- View accounting log

For each sub-task, we assume there is one or methods that must be executed in order to perform the task. Each method may have an access control list (ACL) that defines type of access is required to execute the method. The Read/Write, Read-only, and None access is no longer sufficient. We must define task-based roles for different types of managed-object classes. Using "Manage Printer" as an example, we define the following roles:

- *Senior Administrator* – Reset print queue, Define new printer, View print queue, Delete job from queue, Re-order print queue, and Delete printer
- *Junior Administrator* – Reset print queue, View print queue, Delete job from queue, and Re-order print queue
- *Manager* – View accounting log
- *None* – No access

Some tasks (such as Reset print queue) may have more than one role associated with it. Our administrator task matrix now looks like:

	Printers	Users	File Systems
Jamie	Sr.	Sr.	None
Larry	Jr.	None	Sr.
Shoe	None	Sr.	None

Note how you can mix and match roles between administrators. The next step is to add locale. Keep in mind this is a distributed application,

so you don't necessarily want to give someone the ability to manage all the printers on your network. It makes more sense to delegate administration of a subset of the printers to staff in the department that owns the printers or staff located physically near the printers. Now we expand the printer portion of the task matrix:

Who	What	Where
Jamie	Sr.	Bldg. 6
	Jr.	GLOBAL
Larry	Jr.	Bldg. 5
	None	GLOBAL
Shoe	Sr.	7th floor
	None	GLOBAL

The resources designated by the "Where" column are arbitrary. The administration software must give you the flexibility to define a collection of resources and designate them as "Bldg. 6" or "7th floor". You should not have to be bound by NIS domains or DNS sub-nets since those are very often not the same as the administrative boundaries. You may also want to group resources residing on the same computer system into one or more locales. The GLOBAL locale indicates the role of the administrator for all other locales besides the ones specified.

The last step is how these tasks are performed. For user management, you may want to create home directories in a different location depending on the locale of the user. The task matrix can be expanded to indicate how a task is to be performed:

Who	What	Where	How
Jamie	Sr.	Bldg. 6	/home/<username>
	Sr.	Bldg. 7	/user/home/<username>

The "How" is tied to the "Where", not the role of the administrator. This feature gives you the flexibility to define how a task is performed for a specific set of resources which can even exist on the same host.

Delegation and Authentication

There are many different ways this type of role-based administration system could be implemented. The important features to look for are flexibility and secure authentication. You want to make sure the system will make things easier for you and allow you to delegate tasks to less experienced system administrators. Most of us have created more user accounts than we care to admit and are delighted to turn that role over to someone else. Secure authentication (such as Kerberos) is important because you don't want other users to masquerade as the users who are authorized to perform system administration tasks.

By delegating some of the tasks through the roles we defined above, we aren't giving total control of the system to the different administrators since the root password is no longer required. Each administrator can have as big (or small) a role as you define and you are free to focus on more difficult tasks.

Real-life example

At Tivoli, I have gained experiencing in using a system administration application similar to the one describe in this paper. The Tivoli task matrix for creating users looks similar to:

Who	What	Where
Jamie	Sr.	7th floor
	Jr.	GLOBAL
Larry	Jr.	6th floor
	None	GLOBAL

Tivoli refers to the locales as policy regions which is an arbitrary grouping of resources your can manage with the Tivoli software. The Tivoli software defines 4 roles: Senior, Admin, User, and None which map into the 4 roles we defined earlier: Sr. Administrator, Jr. Administrator, Manager, and None. Administrators may have different roles in different policy regions and you can define what type of resources each policy region may manage (users, printers, etc.)

My experience has been that 4 roles is not enough. You need to be able to define an arbitrary number of roles depending on the task and the locale. At some locales, you may want to allow a certain user to reset a print queue, not not perform any of the other tasks associated with managing printers. Referring to the roles we defined for administering printers, the junior administrator was allowed to: Reset print queue, View print queue, Delete job from queue, and Re-order print queue. Ideally, you would like to be able to define the new role of "local administrator" who may only re-set the printer queue. Fortunately, this is not a hard problem as the ACL are already built into the objects. You can simply add a new role to the ACL. The only software changes required are to those portions of the software that allow you to define the roles themselves.

References

S. Garkinkel and G. Spafford, *Practical UNIX Security*, O'Reilly and Associates, 1991.

D. McNutt, ''Role-based System Administration'', *SuperUser*, May, 1993.

B. Lampson, ''Authentication in Distributed Systems: Theory and Practice'', *ACM Transactions on Computer Systems*, November 1992.

J. Steiner, et. al., ''Kerberos: An Authentication Service for Open Network Systems'', *Usenix Winter 1989 Conference Proceedings*.

T. Smith, ''An Object-Oriented Approach to UNIX Systems Management'', *SANS Conference Proceedings*, Spring, 1992.

Author Information

Dinah McNutt is a System Administration Consultant for Tivoli Systems where she works with customers of Tivoli helping them with system administration problems and customization of Tivoli's system administration software. She has been doing system administration for over 8 years and has written technical articles on the subject for SunExpert Magazine, RS/Magazine, and the X Resource Journal. Ms. McNutt currently writes the Daemons and Dragons column for UNIX Review magazine. She can be reached at dinah@tivoli.com.