RBAC in UNIX Administration

Glenn Faden
Sun Microsystems
gfaden@sun.com

Abstract

This paper describes an implementation of RBAC for UNIX systems in which roles are used as an alternative to the traditional superuser. Roles are special shared accounts which must be formally assumed by authorized users. Each role has a full set of credentials so that it can be authenticated and authorized by existing administrative services. Rather than providing for hierarchical roles, the permissions associated with roles are expressed hierarchically using execution profiles. Extensible attributes for users, roles, and permissions are maintained in distributed databases which can support multiple security policies simultaneously.

1 Introduction

In traditional UNIX systems, the root user is known as the *superuser*, and is exempt from all policy enforcement. The problem with this approach is not just that root is so powerful; it is that everyone else is so weak. Root access is required to perform almost all aspects of administration. There is no hierarchy of privileged operations, no separation of powers, nor the ability to delegate any of the powers to others. There is a mismatch between what is necessary and what is sufficient with respect to access control. For example, setting the system date requires root access, which turn, provides full access to the system.

Role-Based Access Control (RBAC) can be used to partition some of the superuser's powers into a set of discrete roles. This is not the same as actually restricting the power of root, it is parcelling out certain capabilities to others. With RBAC, permissions are assigned to roles and roles are assigned to users[1], where users correspond to real people and roles are associated with functional responsibilities. This is intended to reduce the cost of administration by avoiding repetitive assignments. While assigning permissions to roles works well in many situations, it presents problems for bootstrapping an operating system and administering legacy systems.

RBAC '99 10/99 Fairfax, VA, USA

In this paper we describe a more flexible model for assigning permissions; not only can permissions be assigned to roles, but also to users. This flexibility allows users of the system to be able to perform certain privileged operations without role assumption, while still offering the benefits of other RBAC systems. The RBAC model described in this paper has been prototyped in versions of Solaris™ and Trusted Solaris™, for release in future versions of those products. Trusted Solaris, which fully implement the principle of least privilege, replaces all of the superuser checks throughout the kernel and in certain utilities with checks for fine-grained privileges. RBAC can be used on systems with either the traditional superuser implementation or a privileged based implementation.

2 Roles As Subjects

In some RBAC systems, such as [2] roles are implemented as UNIX groups, and in other systems they are not tied to any existing UNIX concept[3]. In the system described in this paper, roles and users are both types of UNIX accounts. For RBAC to satisfy the requirements for UNIX administration, roles must be authenticated principals. This approach is based on a number of factors relating to authentication, discretionary access, and revocation. Administrative data is typically protected using discretionary access control (DAC), and on Trusted Solaris, mandatory access control (MAC), as well.

For example, consider the Network Information Service, NIS+, the primary repository of administrative data on many UNIX systems. In NIS+, access to columns in a database is based on the authenticated network name of the requesting principal; UNIX groups are not considered in access control. If roles were not principals, then the credentials of normal users would be used for database administration and users would have to configured as NIS+ administrators. This defeats the purpose of using RBAC since membership in roles may change as users come and go. The discretionary access settings for administrative data should remain constant.

There are many other advantages to treating roles as special shared accounts. The essential reason is that the existing mechanisms of UNIX can be used for roles without changes to the kernel or to UNIX semantics. By extending the system's Pluggable Authentication Module (PAM) to recognize role accounts (see Section 7, "Role Assumption," on page 4),, attempts to use roles as primary logins or without authorization can be prevented. One interesting consideration is that when the superuser is itself identified as a role, only users who have been assigned the root role can become the superuser, even if they know the root password.

Another advantage is that principals can have extended attributes such as clearances. In systems that support MAC through the use of labels, administrative data can be protected by assigning labels to files which are only available to principals acting in a role. For example, in Trusted Solaris, all administrative data is labeled with one of two administrative labels that are reserved for roles. MAC offers stronger protection than simply relying on permission bits.

3 Authorizations

The terms permission, privilege, and authorization are frequently used interchangeably because they apply to a variety of operating systems and services. For example, [4] describes how RBAC is used to configure a Web server, and [5] describes its application in databases. In this paper, the terms have a UNIX-oriented definition. In this paper we use these terms to mean different things, and it will help to understand the context of each term.

A *permission* is a generic term which is used to describe a transaction that a user is permitted to do through the execution of a program.

A *privilege* is an attribute of a program (a process attribute) run by a user which is used to override a standard security policy. In traditional UNIX, the term *privileged user* is often used to refer to the superuser because its user ID enables it to override any kernel policy. However, in systems that implement the principle of least privilege in the kernel, an arbitrary number of fine grained privileges can be associated with a process, each of which is used to override a specific policy.

An *authorization* is a right assigned to a user or a role that is used to grant access to an otherwise restricted function. Authorizations are also fine-grained, like privileges, but they are not directly associated with programs. Instead, they are looked up in a database based on a the identity of the user or role. Privilege checks are typically done in the kernel, while authorization checks are done in applications.

3.1 Authorization Hierarchy

An *authorization name* is a unique string that identifies the organization that created the authorization and the functionality it controls. Following the Java™ convention, the hierarchical components of an authorization are separated by dots (.), starting with the reverse order Internet domain of the creating organization, and ending with the specific function within a class of authorizations.

An asterisk is used as a wild card to indicate all authorizations in a class. When a user authorization check is made, the authorization is compared against the explicitly assigned authorization and any wild card entries covering the class (or superclass) of the authorization. For example,

```
solaris.role.*
```

covers the authorizations:

```
solaris.role.delegate,
solaris.role.assign, and
solaris.role.write.
```

3.2 Delegation

When the name of an authorization ends with the reserved word `grant`, the authorization is used to support fine-grained delegation. Users and roles with appropriate grant authorizations can delegate some of their authorizations to others. To delegate an authorization, the user needs to have both the authorization itself and an associated grant authorization which covers it. In addition, the user or role must be authorized in the same administrative domain that the assignment is made.

Authorizations are the key to providing separation of powers and delegation. The databases that are used to maintain attributes of users and roles are controlled by trusted applications that interpret authorizations before making updates. Neither users nor roles are permitted to change the databases directly. Trusted applications, which enforce the system policy, restrict inappropriate updates from taking place. The principal making the request must be authenticated and authorized in the same administrative domain in which the data is maintained.

For practical reasons it is often necessary to have a chief security officer who has all authorizations. This is expressed by assigning the two authorizations

```
solaris.*
```

and

```
solaris.grant.
```

The chief security officer can use these authorizations to delegate powers to other users or roles.

4 Permission Sets

Traditionally in UNIX, trusted applications are assigned a setuid-to-root attribute in their filesystem which gives them the effective user ID of root when they are executed. The power to run these programs is therefore granted to all users of the system.

In some cases, it is preferable to restrict these permissions to specific users or roles. To facilitate the management of these permissions, they are bundled into *execution profiles*. An execution profile is an enumeration of the principal's authorizations and any special process attributes, such as effective user and group IDs, associated with trusted executable objects. These profiles are uniquely named and are stored in a database for retrieval by profile name, username, and executable entity.

4.1 Execution Profiles

Execution profiles are distinct from roles in that they are not principals and do not have entries in account-oriented databases. They are simply collections of permissions that can be assigned as a single entity. Both users and roles may have execution profiles assigned to them. The profile(s) assigned to a user specify the initial set of permissions that a user is granted (without assuming any role), while the profiles assigned to a role replace the user's set upon role assumption.

Execution profiles can contain lists of authorizations and lists of executable entities, such as UNIX commands, *CDE actions*, or *Java codebases*. A CDE action is an executable object in the Common Desktop Environment for UNIX systems. A Java codebase is the pathname to a set of Java class files.

Attributes can be associated with each listed executable, which are interpreted by an appropriate interpreter for each executable type. For UNIX commands and CDE actions, the attributes correspond to the process attributes that are set when the program is run. These include the real and effective user and group IDs for UNIX systems. Trusted Solaris uses additional attributes, such as the inheritable privilege set, extended process attribute flags, sensitivity label, and clearance. Each entry contains a policy value which indicates the variant of Solaris for which the process attributes apply.

For UNIX commands, these attributes are interpreted by a profile execution program, named `pfexec`, which executes the specified command with the process attributes specified in the profile. The standard UNIX shells, `sh`, `csh`, and `ksh` have been modified to invoke `pfexec` for profile-based execution.

For CDE actions, the Desktop Services Library is modified to look up the execution attributes associated with CDE actions in execution profiles.

For Java codebases, the permissions correspond to the Java Authentication and Authorization Service (JAAS) [6] Permission class and are interpreted by a modified JAAS Security Manager.

Note that the permissions can only be supplied by programs that are able to grant them. For this reason, the `pfexec` binary is a setuid-to-root program in Solaris, and has all privileges in Trusted Solaris. However, it does not need to trust the calling process, and relies on trusted database queries to determine the appropriate attributes for a execution.

For Java, the Java Virtual Machine (JVM) must be running as root or as a privileged process to interpret permissions.

4.2 Profile Assignment

One or more execution profiles may be assigned to a user or role. The authorizations in all the profiles are cumulative, so the order of the profiles assigned to a user or role does not affect authorization checks.

However, for execution attributes, the order of the profile list is significant since the attributes are determined by the first matching command or action, and are not combined. For example, the command `/usr/bin/date` may be specified in one profile with an effective user ID of `root`, but in another profile the same command is specified to run as a normal user. Therefore, the most specific and powerful profiles should be listed first, followed by subordinate profiles and any wild card entries.

To reduce the administrative burden, profiles can be nested in a hierarchical manner. Since profiles may contain profiles, the administrator can implicitly assign any number of profiles to a user or role with a single profile assignment. However, this is equivalent to enumerating the profiles in a single list, so the hierarchy is just a convenience.

There are two authorizations that control what profiles can be assigned to users or roles. The authorization

```
solaris.profiles.assign
```

is more powerful, and allows any profile to be assigned. The authorization

```
solaris.profiles.delegate
```

restricts the administrator to assigning profiles that are already in the current user's list.

4.3 Profile Creation

Only a principal with the authorization

```
solaris.profiles.create
```

can create or delete a profile. In addition, there are specific authorizations for assigning executables to a profile and for specifying the security attributes of an executable in a profile. However, administrators with the appropriate `grant` authorization suffixes can assign their associated authorizations to profiles as well as to users and roles.

5 Role Creation

Authorized users or roles can create new roles, modify their attributes, and delete them. The authorization for creation and deletion is

```
solaris.role.write.
```

Roles are created using similar tools to those that are used to create users. Some of the attributes that roles share with users require specific authorizations for administration. For example, the assignment of the role's password requires the authorization

```
solaris.usermgr.passwd.
```

In addition, there are some unique attributes of roles.

Roles can only be assigned to users, not to other roles. *Cardinality* is an attribute that specifies how many times a role can be either assigned or assumed. *Mutual exclusion* specifies that a separation-of-duty relationship exists between this role and other roles.

One of the issues that has been discussed in the literature[7] is the applicability of static vs. dynamic restrictions. For example cardinality and mutual exclusion can be enforced when roles are assigned, when they are assumed, or both. In practice, dynamic restrictions are not very useful on networked systems, because a role which is authenticated on one system can extend its credentials through single-signon, across the network to other systems. A more effective approach is for the various services to support file locking and concurrency control. Therefore, only static attribute restrictions are well supported.

6 Role Assignment

The rights to create and modify roles does not convey the right to assign them to others. There are two authorizations required for role assignment. The authorization

```
solaris.role.assign
```

is more powerful, and allows any role to be assigned or revoked. The authorization

```
solaris.role.delegate
```

allows a user to assign a role to another user only if the first user is already assigned the role.

Note that granting a role to another user does not give the second user the right to further delegate that role. unless the second user also has

```
solaris.role.delegate.
```

There are no authorizations to override the restrictions of cardinality and mutual exclusion. However, the authorization for creating roles does provide for modifying the cardinality and mutual exclusion restrictions.

7 Role Assumption

Role assumption is the discrete action of activating a role that has been assigned to a user. Since roles are limited to authorized users, the identity of the user must be authenticated before the role assumption can take place. Therefore, roles cannot be used as primary login accounts. The users must first login to the system and then use an appropriate interface to assume a role.

The following figure shows that the profile sets for a user and a role are distinct.

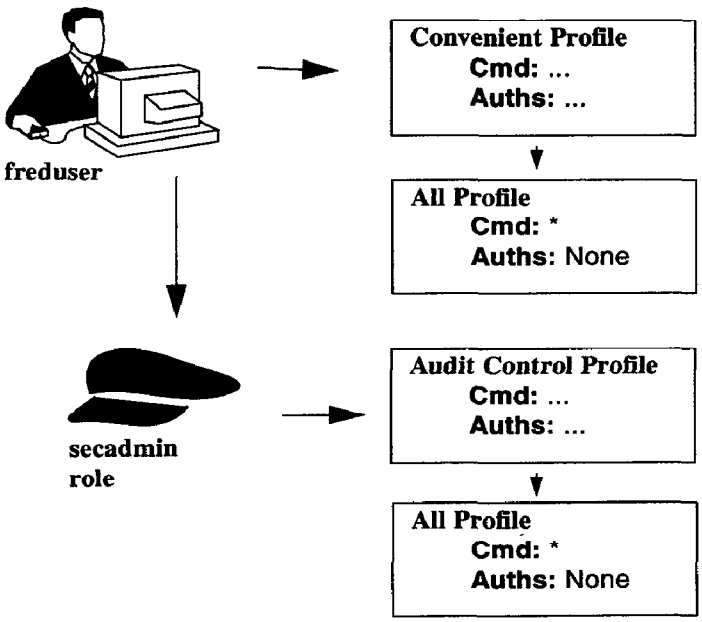

Figure 1 Assuming a role

The simplest method to assume a role is by using the traditional `su` command. Other assumption interfaces are provided in administrative GUIs, such as the CDE Workspace Manager front panel. In order to assume a role, authentication and authorization checks are made. Both checks are implemented using the Pluggable Authentica-

tion Module described in the X/Open Single Signon Option [8]. One or more authentication modules, such as a password authenticator, are called to authenticate the role. If successful, an additional module is called to verify that the role has been assigned to the user who is assuming it, and that any dynamic restrictions, such as cardinality or mutual exclusion, are not violated.

If authentication and authorization are successful, the attributes for the role are set up, and further execution takes place as the role. For purposes of attribution, the audit ID of the user who assumed the role is preserved, but the audit event mask is set to the value assigned to the role.

In order to take advantage of execution profiles, roles are assigned a profile enabled version of one of the standard UNIX shells. These shells restrict execution to the set of commands enumerated in the role's profiles and apply the special attributes identified for each command. Wild card entries can be specified in profiles to indicate all commands in a directory, or simply, all commands.

Role assumption is not a cumulative operation with respect to attributes. An authorization granted to a user is not conveyed to the role when it is assumed. Furthermore, roles are not hierarchical, and cannot assume other roles. Although this may seem at odds with other RBAC systems [9], it provides a consistent set of rules for Solaris administration. The concept of separation-of-duty is more easily understood if roles run in separate environments. For example, in Trusted Solaris, when a role is assumed in the CDE environment, it is isolated in its own CDE workspace, which can only display windows associated with the role.

The notion of role hierarchy conflicts with the straightforward model that is presented by treating roles as accounts. Since all the capabilities of the role are available when it is assumed, there is no question about whether a subordinate role is also active. When roles are accounts, the role ID is passed through normal UNIX inheritance of process attributes. For example, multiple terminal windows can be brought up by a role, each of which share the same role attributes.

8 RBAC Databases

In the Solaris and Trusted Solaris prototypes, the attributes required to support RBAC are maintained in four databases. Each of these databases is implemented using a name service so that databases may be centrally maintained or distributed.

8.1 user_attr

The attributes associated with users and roles are stored in the `user_attr` database. Roles and Users are distinguished by a type attribute. The list of available roles is determined by scanning the database for entries whose type is `role`. The set of attributes for users includes a list of profiles, and a list of roles. The attributes for roles include a list of profiles, the cardinality constraints, and a list of mutually excluded profiles.

8.2 auth_attr

The list of available authorizations and their descriptive attributes are stored in `auth_attr`. The attributes for authorizations include a more user-friendly name, and a reference to its help description file.

8.3 prof_attr

The list of available profiles and their attributes are stored in `prof_attr`. The profile attributes include any authorizations that are associated with the profile, any subordinate profiles, and a reference to its help description file.

8.4 exec_attr

The list of executables that require special execution attributes is stored in `exec_attr`. For each entry, there is a reference to the profile with which it is associated. Other attributes include the type of executable, e.g. UNIX command or CDE action, the fully qualified name of the executable, and the process attributes it will be assigned when executed.

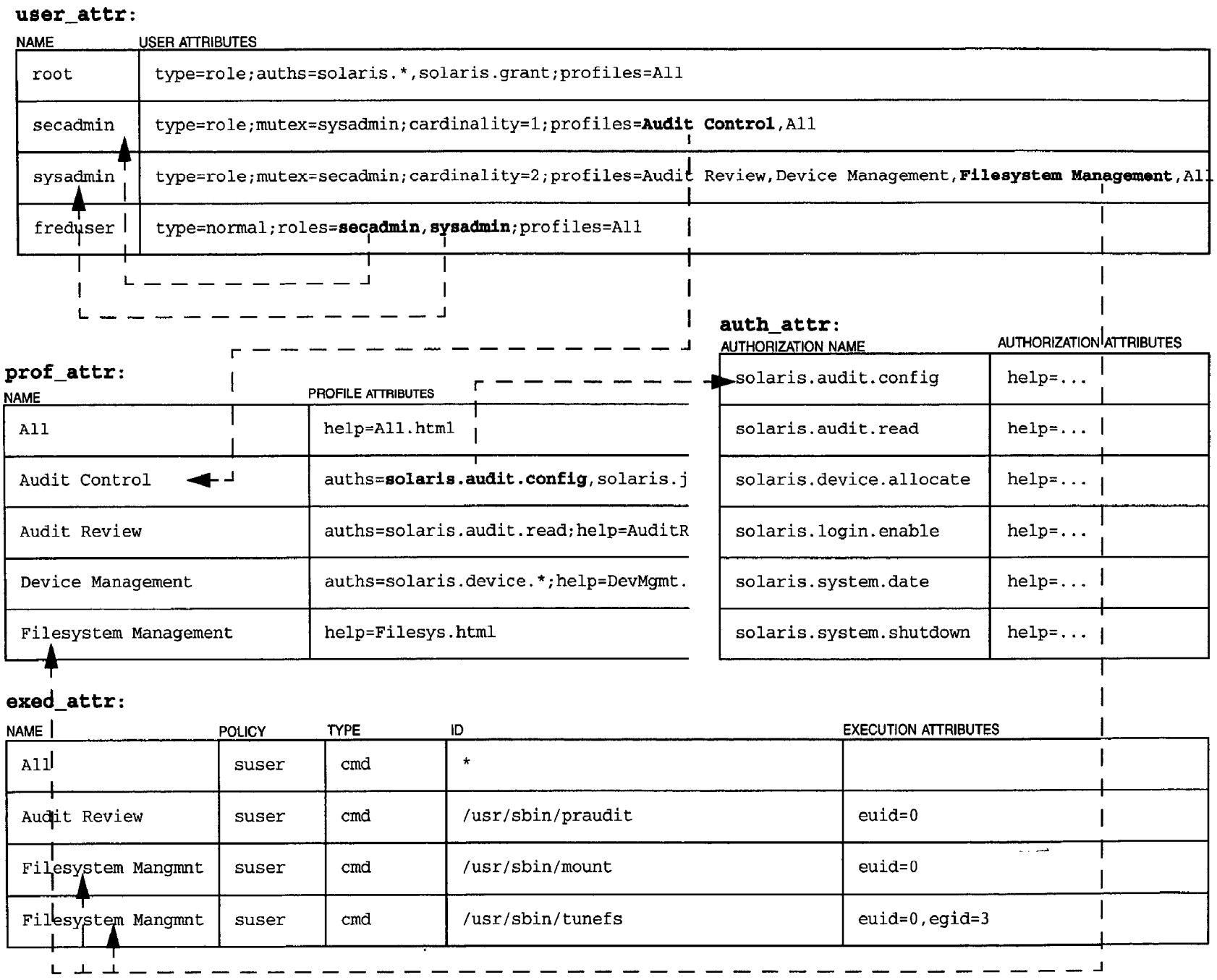

Figure 2 RBAC Database Relationships

Figure 2, RBAC Database Relationships, shows how some of these attributes are related.

9 Conclusion

This RBAC implementation draws on traditional UNIX security as well as newer models for delegation and separation-of-powers. It defines another entity, an execution profile, which is used to manage permissions because the maintenance of permission sets, rather than role attributes is where the biggest administrative difficulties exist. Fine-grained authorization is built into the design of the RBAC databases so that various aspects of their management can be assigned to separate roles.

Treating roles as UNIX principals allows existing applications and interfaces to work with RBAC without requiring that they be rewritten to use new interfaces and databases. Rather than defining a specific server or application that understands and interprets roles, this approach allows a phased evolution of UNIX administrative concepts from the traditional superuser to a model where rights are granted based on what is necessary and sufficient to perform the task at hand.

10 References

[1 *Role-Based Access Control Protection Profile*, Common Criteria, July 30, 1998

[2] *Decentralized Group Hierarchies in UNIX: An Experiment and Lessons Learned*, R. Sandhu & G Ahn, In Proceedings of 21st NIST-NCSC National Information Systems Security Conference, pages 486-502

[3] *Role-Based Access Control (RBAC): Features and Motivations*, David Ferraiolo et al., Computer Security Applications Conference, 1995

[4] *A Role Based Access Control Model and Reference Implementation within a Corporate Internet.* David F. Ferraiolo, John F. Barkley, and D. Richard Kuhn, ACM Transactions on Information Systems Security, Volume 1, Number 2, February 1999, National Institute of Standards and Technology

[5] *Role-Based Access Control In Commercial Database Management Systems*, D.F and D.R. Kuhn, *In Proceedings of 21st NIST-NCSC National Information Systems Security Conference*, pages 503-511

[6] *Java Authentication and Authorization Service*, http://java.sun.com/security/jaas, April 13, 1999.

[7] *Specifying and Managing Role-Based Access Control within a Corporate Intranet*, Ferraiolo, Barkley, 1997, Second ACM Workshop on Role-Based Access Control, 1997

[8] *X/Open Single Sign-on Service (XSSO) - Pluggable Authentication Modules,* August 5, 1998

[9] *Inheritance Properties of Role Hierarchies*, W. Jansen, In Proceedings of 21st NIST-NCSC National Information Systems Security Conference, pages 476-485