Implementing RBAC on a Type Enforced System

John Hoffman
Secure Computing Corporation
2675 Long Lake Road
Roseville, MN 55113
hoffman@securecomputing.com

Abstract

Role Based Access Control (RBAC) has gathered much attention in recent literature. Much of the discussion has focused on theoretical issues, potential features, or on web or security database implementations. This paper describes an implementation of RBAC mechanisms on LOCK6, a secure operating system developed at Secure Computing Corporation. The implementation has the RBAC features necessary to solve the usual problems in our application domain (that of firewall construction) while providing a path to many of the more advanced RBAC features needed by other application domains. Finally we argue RBAC alone is not a sufficient mechanism to produce secure systems, but that an additional lower level mechanism such as Type Enforcement is required.

1 Introduction

Role Based Access Control (RBAC) has recently garnered a fair amount of attention in the literature [3, 9, 10]. These papers discuss a high level policy and the features available in such a policy, without actually discussing how one might implement an RBAC policy. While much work has been done in RBAC as it applies to databases (most current relational DBMS implementations have RBAC features; Oracle and Sybase are two examples) and web servers [1], little has been published on how to implement RBAC on a general purpose operating system. This paper will describe the RBAC policy implemented on LOCK6, a Type Enforced operating system developed at Secure Computing Corporation as part of the Secure Network Server program.

We have found RBAC to be an effective method to aid in the administration of a Type Enforced operating system. Type Enforcement can be viewed as an extra layer of abstraction between the concept of a role and individual permission bits in the operating system. As such, RBAC facilitates system administration and secure system creation. In addition, many of the high level RBAC features are easily implemented through appropriate administration utilities on a Type Enforced system. Finally, we will argue that in order for an RBAC mechanism to support least privilege, the extra layer abstraction supplied by Type Enforcement is necessary.

```
%   Throughout the paper we have
%   stated expressions formally
%   using the PVS specification
%   language.  (See Owre in the
%   bibliography for a reference.)
%   All formalizations appear in
%   this font.  We include these
%   formalizations as an aid to
%   those readers who understand
%   formal notations.  Those
%   unfamiliar with these
%   notations can skip them and
%   not miss any of the important
%   contents of the paper.
%   However, if there is a
%   discrepancy between the
%   English text and the formalisms,
%   the formalisms take precedence.

%   The PVS is only used as a
%   specification tool in this paper;
%   we prove no theorems.  Also, we
%   have made an effort to avoid
%   portions of the language that
%   would be confusing to those not
%   conversant with PVS.  The one
%   term that may require explanation
%   is TYPE+.  "foo : TYPE+"
%   describes a new data type called
%   foo.
```


0-8186-8274-4 $10.00

2 What is RBAC?

RBAC has been developed as a standard access control policy to address the information security needs of civilian, government, and commercial enterprises. In these organizations, disclosure of information is not as important as integrity of information. A natural mechanism to determine who has access to what information is the role of an individual in an organization. Thus, RBAC policies relate users to roles, and roles to operations on a computer system [3, 8]. For example, a system administrator should have different accesses than a system operator.

RBAC policies are generally stated, however, at a very high level of abstraction. One point of this paper is to demonstrate that the abstractions described in an RBAC policy map in a natural way to the control mechanisms available in a Type Enforced system. In addition to show the ease with which such a mapping facilitates an implementation of an RBAC policy.

There are many different flavors of policy that can be supported with varying degrees of sophistication, as Sandhu et.al.[8] have pointed out. Throughout our document we will be referring to Ferraiolo's RBAC policy [3] as a canonical high level RBAC policy. In his policy, there are users, roles, subjects, and operations. How these entities are related is discussed further in section 4.

Sandhu [9] provides a more complete overview of RBAC and describes its advantages more completely. Before delving further into RBAC, we describe Type Enforcement.

3 What is Type Enforcement?

Type Enforcement is a low level mandatory access control mechanism that restricts the accesses a subject can have to objects through the use of domain labels on subjects and type labels on objects. It is important to note a distinction between Type Enforcement and RBAC. RBAC ties users to roles and describes how a role limits the operations available to a user. Type Enforcement ties subjects to domains and describes how a domain limits the operations available to a subject. Type Enforcement and its applications have been described in many papers [2, 4, 5, 6].

3.1 Basic definitions

In Type Enforcement each subject on the system is assigned a domain

```
Subject : TYPE+
Domain  : TYPE+

subject_domain  :
   FUNCTION[Subject -> Domain]
```

and every object on the system is assigned a type.

```
Object  : TYPE+
TEType  : TYPE+

object_type     :
   FUNCTION[Object -> TEType]
```

The accesses permitted to a subject for an object will depend on the subject's domain and the object's type. Before formalizing this concept we provide some more background.

3.2 Example

Secure Computing Corporation's current Type Enforced operating system is known as LOCK6, which has an object oriented design. Objects have an interface; the only means by which a subject can access an object is through the interface. The methods of objects are implemented by an "animator" subject that generally resides in an address space separate from the client subjects. Objects on LOCK6 include, but are not limited to, standard operating system entities such as directories, files, pipes, sockets, devices, and anonymous memory objects.

A running example throughout this paper will be the development of a message guard (see figure 1). A message guard is a firewall application that filters mail passing between two networks (for example, a corporate intranet and the Internet). A message guard consists of:

- two message transfer agents (MTAs) (e.g. X.400 MTA's, or sendmail), each of which can communicate with exactly one network
- two filter pipelines, one for internal to external mail flow, the other for the external to internal mail flow. Each filter pipeline consists of various filter subjects that filter mail.

Type Enforcement is a means to categorize objects and subjects on the system as well as their interrelationships. As an example, all files on a system containing mail messages originating from the internal network would be of type Mail_i, and the message transfer agent for the internal network would have a domain of MTA_I.

3.3 Complete definition

The Type Enforcement policy can now be stated precisely. There is a collection of accesses possible on the system.

```
Access : TYPE+
```

For every possible (Domain, Type) pair, there is a set of permitted accesses that subjects operating in that domain can perform on objects of the type. This table of permissions is called the Type Enforcement database.

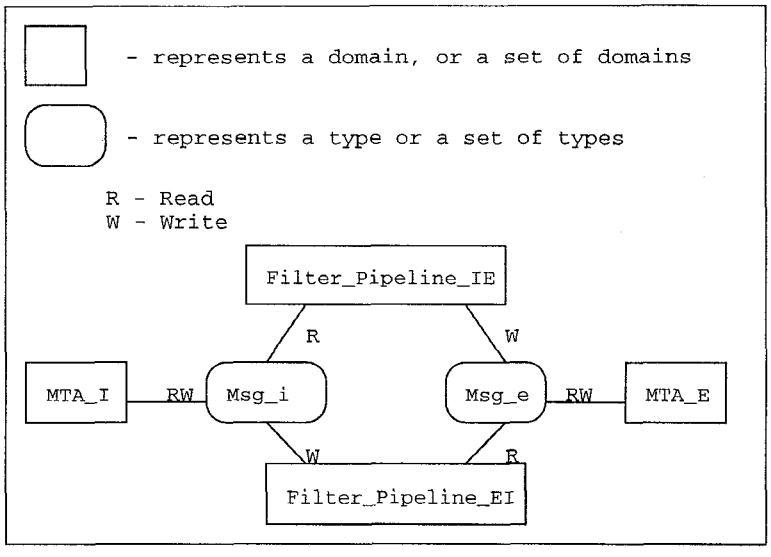

Figure 1. Example domain/type structure with associated permissions

```
TE_database :
   FUNCTION[[Domain, TEType]
       -> setof[Access]]
```

There is a subject associated with every change (or operation) on the system.

```
Operation : TYPE+
operation_client :
   FUNCTION[Operation -> Subject]
```

During every operation of the system, some objects are accessed.

```
operation_objects_accessed :
   FUNCTION[Operation
      -> setof[[Object,Access]]]

% For an operation op,
% operation_objects_accessed(op)
% is a set of ordered pairs (o,a)
% such that access "a" occurred
% to object "o".
```

The Type Enforcement policy says that any object accesses that occur during an operation are consistent with the Type Enforcement database. As an example, if a message transfer agent deletes a file, then the message transfer agent had permission to delete the file.

```
TE_policy : THEOREM
   forall (op : Operation),
          (obj : Object),
          (acc : Access) :

   member((obj, acc),
       operation_objects_accessed(op))
  IMPLIES
   member(acc,
          TE_database(
              subject_domain(
                operation_client(op)),
              object_type(obj)))
```

For this policy to work on a real system, it requires some connection to users. Roles provide this connection.

4 Implementing RBAC using Type Enforcement

In a typical RBAC system/policy [3] there are collections of roles and users.

```
Role   : TYPE+
User   : TYPE+
```

Each subject on the system has a role and a user associated with the subject.

```
subject_role : [Subject -> Role]
subject_user : [Subject -> User]
```

The NIST policy [3] (which we are using as a basis for a standard RBAC policy) assigns a collection of roles to each subject, which is different from this approach of a single

role for each subject. We will discuss this difference in more detail in section 6.

Each user is assigned a set of roles. These are the roles in which the user is authorized to operate. A system administrator determines this mapping on the system.

```
user_roles : [User -> setof[Role]]
```

Since every subject has a role and a user, and every user has a set of roles, these relationships should be consistent. That is, for every subject, the role of the subject must be an authorized role for the subject's user.

```
subject_user_role_consistent
        : THEOREM
forall (sbj : Subject) :
member(subject_role(sbj),
       user_roles(subject_user(sbj)))
```

At this point a typical RBAC policy connects operations and roles, it requires each operation on the system performed by a subject is an operation appropriate for the role of the subject. However our TE policy connects operations and domains. Essentially we have introduced a new layer of abstraction between that of operations and roles. Thus, in our policy, and implementation, we connect roles to domains.

To connect roles to domains, we associate a set of domains with each role. This association of a set of domains to a role is stored on the system in the security databases. The contents of these databases are determined by the system security policy.

```
role_domains : [Role -> setof[Domain]]
```

Again, we are confronted with a consistency issue. Every subject has a role and a domain, and all roles have a set of domains. Thus, the domain of a subject should be in the set of domains authorized for the role.

```
subject_domain_role_consistent
        : THEOREM
forall (sbj : Subject) :
member(
   subject_domain(sbj),
   role_domains(subject_role(sbj)))
```

Given this approach, all access to objects by subjects is determined by the Type Enforcement policy. This is important because Type Enforcement permissions are easily determined by what duties a subject performs.

To summarize this RBAC implementation, a subject has an associated user, role, and domain. A subject's domain determines the accesses it has to an object. The subject's user, role and domain must all be consistent with the security databases which specify what roles are allowed to a user and what domains a role is allowed to operate.

5 Differences between this RBAC implementation and some of the more general RBAC policies

Allowing a subject to operate in only one role at a time is a bit different from the approach described in the NIST policy [3], where a subject can operate in multiple roles simultaneously. We believe a subject operating in a single role at a time is more desirable than allowing a subject to operate in all its roles, because the semantics of the operations of a subject can depend on the role. As behavior of a subject can depend on its set of current roles, there is extra responsibility placed on the user of the system, because the user must know what roles he is operating in and he must understand the behavior of a subject given the current set of roles. This is an unnecessary burden. We develop our systems to be easily administered by people with limited training, thus everything must be kept as conceptually simple as possible.

To clarify this point, consider a subject status utility on a message guard. A utility that allows the user to examine subjects executing on the system and to modify their behavior; either by destroying them or sending them signals. This subject should operate differently for different roles. The system administrator should be allowed to examine and manipulate all subjects. However, the message administrator needs only to access mail subjects. Thus, although the same executable is being used, its behavior should depend on the role in which it is operating. Moreover it is desired to limit the functionality available to a user who can operate in both roles but who is interested only in the message guard. This could be described as user least privilege. It is useful in reducing mistakes made by a user. A mistake made in the system administrator version of the utility could crash the system, while a mistake in the message administrator version would merely shutdown the message guard.

6 Problems solved by this approach to RBAC

Generally the literature states that the accesses a subject has to an object should depend only on the roles of the subject. (Note the set of roles for a subject is considered to be an appropriate subset of the set of allowable roles for the user of the subject.) This is not sufficient for most applications. For any system that is running untrusted software, the accesses an untrusted subject has to an object should depend on more than a role. (In this context trusted software is software that is trusted to not abuse or otherwise misuse any of the privileges or permissions it is given.) It is necessary that untrusted software be limited to access only the objects that are required for it to do its task (i.e., to satisfy least privilege). The accesses associated for a given subject are likely a small subset of all the accesses necessary for the role to perform its duty.

Of course these kinds of restrictions could be accomplished through roles. One could define many different roles, one role for each kind of subject. But doing this violates the intuitive concept of a role, something that corresponds to a role in an organization. In addition, the user now must be aware of all of these micro roles, and must know why they are there. It makes using the system more complicated, something that should be avoided.

Returning to the guard model, a message administrator needs to be able to run several diagnostic programs for the MTA. These diagnostic programs as well as the MTAs are large bodies of ported, complicated code that should not have to be "trusted." Another duty of the message administrator is to modify the filter pipeline configuration files. It is imperative that the large bodies of untrusted code not be able to modify these configuration files. This is easily accomplished by having the MTA diagnostic subjects run in MTA_DIAGNOSTIC domain (note this is not part of the figure 1), and make sure that this domain has no modify access to files of type FILTER_CONFIGURATION. Thus the lower level abstraction of domain facilitates a finer layer of control, necessary to separate trusted and untrusted software.

In this example we have argued that it is desirable for a subject's accesses to objects to depend upon more than just its role. Thus, we feel the added refinement of domains (or something similar) is necessary for implementing an RBAC policy on a general purpose operating system.

Another advantage of our approach to RBAC is the clarity it brings to our top down development process. Once roles are identified, along with their associated duties, a collection of subjects and domains necessary to perform those duties is easily derived.

In the message guard example, the message administrator configures message transfer agents (MTA), configures the filter pipeline configuration, and starts/stops the MTA's and filter pipeline. These duties suggest several domains: a MTA configuration domain, a filter pipeline configuration domain, and an MTA/filter status domain. Once these domains have been decided upon, it is again straightforward to determine the appropriate separate types for the files that these domains must access. A type is needed for filter configuration files, for MTA configuration files, and a type for filter and MTA status files. The point being that the accesses a domain need to a type are not so much determined by the role they operate in, but by the specific sub duties of the role the subject is to perform.

7 Higher level RBAC properties

Many papers on RBAC discuss other features of RBAC that users would like or need. Not every RBAC feature is implemented in LOCK6, the missing features include, role hierarchies, separation of duty, membership limits. The mechanisms of LOCK6 support most of these features, all that is needed is the addition of appropriate interfaces or administrative applications. We discuss the interfaces needed to fully implement these features. We demonstrate in this section that LOCK6's approach is robust enough to easily support many of the additional features needed by other application domains. The difficult task of getting the low level semantics of RBAC and Type Enforcement into the operating system is finished. Readers interested in more detail about these features are directed to [3, 8].

7.1 Role hierarchies

A role hierarchy is a partial order on roles. Generally, if a user is permitted in a role `foo`, she gains all the privileges to all the roles that are dominated by `foo` in the hierarchy. In a Type Enforcement implementation of RBAC, the dominates relation of the partial order is determined by set containment. A role `foo` dominates a role `bar` if the set of domains for `foo` is a superset of the domains for `bar`.

7.2 Separation of duty

Separation of duty ensures that different individuals carry out certain collections of duties. There are two commonly discussed means by which this separation can be assured; static and dynamic separation of duty.

7.2.1 Static separation of duty

Static separation of duty ensures that different individuals carry out certain collections of duties by assigning these duties to different roles, and then limiting the roles any one user can belong to. As an example, consider a user who can make a purchase order and write a check. This should not be allowed. Thus, the role that creates purchase orders and the role that writes checks are mutually exclusive.

This is a restriction on the collection of roles associated with a user. On LOCK6 the correspondence between users and roles is stored in files of type USER_ROLE_TABLE_TYPE. Static separation of duty restrictions must then be enforced by any subjects that modify these files. The determination of what subjects modify files of this type is easily made by examining the Type Enforcement databases. Any subject that has modify access to files of this type must be assured to maintain this restriction.

7.2.2 Dynamic separation of duty

Dynamic separation of duty restricts the set of roles in which a user can actively be operating. For example a user cannot simultaneously operate as a cashier and as a supervisor. The implementation of LOCK6 is such that the files of type

USER_ROLE_TABLE_TYPE define the set of roles in which a user is currently acting. Currently this is all of a user's ROLES. To change this situation, a new application would need to be written that would allow users to specify the set of roles in which they would like to operate. Adding this feature would require no change to the operating system.

7.3 Membership limits

7.3.1 Static membership limits

Static membership limits restrict the number of users that can be authorized for a particular role. These limits need to be enforced any time the USER_ROLE_TABLE is modified. Thus, all subjects with modify permission to the USER_ROLE_TABLE type must be trusted to perform these modifications correctly. This is very much like static separation of duty.

7.3.2 Dynamic membership limits

Dynamic membership limits require the system to keep track of the roles in which a user is actively participating. This is a simple computation based on the state of the system. Implementing this requires changes as described in the sections on dynamic separation of duty and in static membership limits.

7.4 Other areas for investigation

Currently the LOCK6 system is configured with a fixed set of roles. This is from a desire to ship systems that will never get into an insecure state due to configuration errors by users. However, as our application bases expand, tools to allow an administrator to modify and/or create roles will become necessary. Again, this would require just an application with an appropriate user interface that runs in a special domain allowed to modify the object containing the role-domain correspondence. It requires no modifications to the underlying operating system.

8 Conclusion

We have found RBAC to be an effective method to aid the administration of Type Enforced operating systems. It can be viewed as an extra layer of abstraction from the concept of a domain, and as such it facilitates system administration and system creation. In addition, Type Enforcement is a straight forward method of dealing with some of the problems of implementing an effective RBAC policy. It provides a lower level of abstraction that facilitates secure system design and encourages well formed transactions of data. Finally, many of the higher level RBAC concepts are easily implemented through appropriate administration utilities on a Type Enforced system.

References

[1] J. F. Barkley, A. V. Cincotta, D. F. Ferraiolo, S. Gavrilla, and D. R. Kuhn. Role based access control for the world wide web. http://hissa.ncsl.nist.gov/rbac/rbacweb/paper.ps, April 1997.

[2] W. Boebert and R. Kain. "A Practical Alternative to Hierarchical Integrity Policies". In *Proceedings of the 8th National Computer Security Conference*, pages 18–27, October 1985.

[3] D. Ferraiolo, J. Cugini, and R. Kuhn. Role-based access control (RBAC): Features and motivations. In *Computer Security Applications Conference*, 1995. http://hissa.ncsl.nist.gov/rbac/newpaper/rbac.html.

[4] P. Greve, J. Hoffman, and R. Smith. Using type enforcement to assure a configurable guard. In *Proceedings of 13th Annual Computer Security Applications Conference*, December 1997.

[5] J. Haigh. "Modeling Database Security Requirements". In C. Landwehr, editor, *Database Security: Status and Prospects*. North-Holland, 1988.

[6] R. O'Brien and C. Rogers. Developing applications on LOCK. In *Proceedings of 14th National Computer Security Conference*, pages 96–106, October 1991.

[7] S. Owre, S. Rajan, J. Rushby, N. Shankar, and M. Srivas. PVS: Combining specification, proof checking, and model checking. In R. Alur and T. A. Henzinger, editors, *Computer-Aided Verification, CAV '96*, number 1102 in Lecture Notes in Computer Science, pages 411–414, New Brunswick, NJ, July/August 1996. Springer-Verlag.

[8] R. Sandhu, E. Coyne, H. Feinstein, and C. Youman. Role-based access control: A multi-dimensional view. In *Tenth Annual Computer Security Applications Conference*. IEEE Computer Society Press, December 1994.

[9] R. Sandhu, E. Coyne, H. Feinstein, and C. Youman. Role-based access control models. *IEEE Computer*, 29(2):38–47, February 1996.

[10] C. Smith, E. Coyne, C. Youman, and S. Ganta. A marketing survey of civil federal government organizations to determine the need for a role-based access control (RBAC) security product. Technical report, NIST and Seta Corporation, July 1996.